

RAIL

[Developing with RAIL](#)

[Release Notes](#)

[RAIL SDK](#)

[Getting Started](#)

[Overview](#)

[Quick Start Guide \(PDF\)](#)

[EFR32 Migration Guide for Proprietary Applications](#)

[Introduction](#)

[Minor Differences Between EFR32 Generations](#)

[RF Sense On EFR32xG22](#)

[EM1P](#)

[RAIL Fundamentals](#)

[Introduction](#)

[RAIL Overview](#)

[RAIL Library](#)

[RAIL Features](#)

[Major Updates to RAIL](#)

[Next Steps](#)

[Developer's Guide](#)

[Overview](#)

[Using RAIL for IEEE 802.15.4 Applications with EFR32](#)

[Overview](#)

[Configure 802.15.4 PHYs In Rail](#)

[802.15.4 HW Accelerated MAC Features](#)

[Examples](#)

[Power Amplifier Power Conversion Functions in RAIL 2.x](#)

[Introduction](#)

[EFR32 Power Amplifiers And RAIL](#)

[Generating Conversion Data](#)

[Alternative Methods To Convert Between Power Levels And DBm Output Power](#)

[Detailed Timing Test Results for RAIL \(PDF\)](#)

[Direct Mode](#)

[PHY Development](#)

[Overview](#)

[EFR32 Radio Configurator Guide \(PDF\)](#)

[EFR32 Series 1 Long Range Configuration Reference \(PDF\)](#)

[EFR32 Series 2 Long Range Configuration Reference \(PDF\)](#)

[EFR32 RF Evaluation Guide](#)

[Introduction](#)

[Running Simplicity Studio](#)

[Basics of the RAILtest Application](#)

[Transmission Test](#)

[Reception Test](#)

[Silicon Labs RAIL API Reference Guide](#)

[RAIL API](#)

[Antenna Control](#)

[RAIL_AntennaConfig_t](#)

[ant0PinEn](#)

[ant1PinEn](#)

[defaultPath](#)

[ant0Port](#)

[ant0Pin](#)

[ant1Loc](#)

[ant1Port](#)

[ant1Pin](#)

[RAIL_AntennaSel_t](#)

[RAIL_ConfigAntenna](#)

[RAIL_GetRfPath](#)

[Assertions](#)

[RAIL_AssertErrorCodes_t](#)

[RAILCb_AssertFailed](#)

[Auto-Ack](#)

[RAIL_AutoAckConfig_t](#)

[enable](#)

[ackTimeout](#)

[rxTransitions](#)

[txTransitions](#)

[RAIL_ConfigAutoAck](#)

[RAIL_IsAutoAckEnabled](#)

[RAIL_WriteAutoAckFifo](#)

[RAIL_GetAutoAckFifo](#)

[RAIL_PauseRxAutoAck](#)

[RAIL_IsRxAutoAckPaused](#)

[RAIL_PauseTxAutoAck](#)

[RAIL_IsTxAutoAckPaused](#)

[RAIL_UseTxFifoForAutoAck](#)

[RAIL_CancelAutoAck](#)

[RAIL_IsAutoAckWaitingForAck](#)

[Calibration](#)

[RAIL_TxIrCalValues_t](#)

[dcOffsetIQ](#)

[phiEpsilon](#)

RAIL_IrCalValues_t
 rxIrCalValues
 txIrCalValues
EFR32xG2x
 RAIL_ChannelConfigEntryAttr
 calValues
Slxx3xx
 RAIL_ChannelConfigEntryAttr
 calValues
RAIL_CalMask_t
RAIL_RxIrCalValues_t
RAIL_CalValues_t
RAIL_ConfigCal
RAIL_Calibrate
RAIL_GetPendingCal
RAIL_ApplyIrCalibration
RAIL_ApplyIrCalibrationAlt
RAIL_CalibrateIr
RAIL_CalibrateIrAlt
RAIL_CalibrateTemp
RAIL_CalibrateHFXO
RAIL_EnablePaCal
RAIL_BLE_CalibrateIr
RAIL_IEEE802154_CalibrateIr2p4Ghz
RAIL_IEEE802154_CalibrateIrSubGhz

Chip-Specific

RAIL_EFR32xG2x_Interrupts
 FRC_PRI_IRQHandler
 FRC_IRQHandler
 MODEM_IRQHandler
 RAC_SEQ_IRQHandler
 RAC_RSM_IRQHandler
 BUFC_IRQHandler
 AGC_IRQHandler
 PROTIMER_IRQHandler
 SYNTH_IRQHandler
 RFSENSE_IRQHandler
 PRORTC_IRQHandler
 HOSTMAILBOX_IRQHandler
 RDMAILBOX_IRQHandler
 RFECA0_IRQHandler
 RFECA1_IRQHandler
 RFTIMER_IRQHandler
 SOFTM_IRQHandler

RFLDMA_IRQHandler
SYSRTC_SEQ_IRQHandler
EMUDG_IRQHandler

Data Management

RAIL_DataConfig_t
txSource
rxSource
txMethod
rxMethod
RAIL_TxDataSource_t
RAIL_RxDataSource_t
RAIL_DataMethod_t
RAIL_ConfigData
RAIL_WriteTxFifo
RAIL_SetTxFifo
RAIL_SetTxFifoAlt
RAIL_SetRxFifo
RAILCb_SetupRxFifo
RAIL_ReadRxFifo
RAIL_SetTxFifoThreshold
RAIL_SetRxFifoThreshold
RAIL_GetTxFifoThreshold
RAIL_GetRxFifoThreshold
RAIL_ResetFifo
RAIL_GetRxFifoBytesAvailable
RAIL_GetTxFifoSpaceAvailable

Diagnostic

RAIL_DirectModeConfig_t
syncRx
syncTx
doutPort
doutPin
dclkPort
dclkPin
dinPort
dinPin
doutLoc
dclkLoc
dinLoc
RAIL_VerifyConfig_t
correspondingHandle
nextIndexToVerify
override
cb

RAIL_StreamMode_t
RAIL_FrequencyOffset_t
RAIL_VerifyCallbackPtr_t
RAIL_ConfigDirectMode
RAIL_EnableDirectMode
RAIL_EnableDirectModeAlt
RAIL_GetRadioClockFreqHz
RAIL_SetTune
RAIL_GetTune
RAIL_SetTuneDelta
RAIL_GetTuneDelta
RAIL_GetRxFreqOffset
RAIL_SetFreqOffset
RAIL_StartTxStream
RAIL_StartTxStreamAlt
RAIL_StopTxStream
RAIL_StopInfinitePreambleTx
RAIL_ConfigVerification
RAIL_Verify

Events

RAIL_Events_t
RAIL_ConfigEvents

External Thermistor

RAIL_HFXOThermistorConfig_t
 port
 pin
RAIL_HFXOCompensationConfig_t
 enableCompensation
 zoneTemperatureC
 deltaNominal
 deltaCritical
RAIL_StartThermistorMeasurement
RAIL_GetThermistorImpedance
RAIL_ConvertThermistorImpedance
RAIL_ComputeHFXOPPMError
RAIL_ConfigHFXOThermistor
RAIL_ConfigHFXOCompensation
RAIL_GetHFXOCompensationConfig
RAIL_CompensateHFXO

Features

RAIL_Supports2p4GHzBand
RAIL_SupportsSubGHzBand
RAIL_SupportsDualBand
RAIL_SupportsAddrFilterAddressBitMask

RAIL_SupportsAddrFilterMask
RAIL_SupportsAlternateTxPower
RAIL_SupportsAntennaDiversity
RAIL_SupportsPathDiversity
RAIL_SupportsAuxAdc
RAIL_SupportsChannelHopping
RAIL_SupportsDirectMode
RAIL_SupportsDualSyncWords
RAIL_SupportsTxRepeatStartToStart
RAIL_SupportsVdet
RAIL_SupportsExternalThermistor
RAIL_SupportsHFXOCompensation
RAIL_SupportsMfm
RAIL_SupportsOFDMPA
RAIL_SupportsPrecisionLFRCO
RAIL_SupportsRadioEntropy
RAIL_SupportsRfSenseEnergyDetection
RAIL_SupportsRfSenseSelectiveOok
RAIL_SupportsRssiDetectThreshold
RAIL_SupportsRxDirectModeDataToFifo
RAIL_SupportsRxRawData
RAIL_SupportsSQPhy
RAIL_SupportsTxPowerMode
RAIL_SupportsTxPowerModeAlt
RAIL_SupportsTxToTx
RAIL_SupportsProtocolBLE
RAIL_BLE_Supports1MbpsNonViterbi
RAIL_BLE_Supports1MbpsViterbi
RAIL_BLE_Supports1Mbps
RAIL_BLE_Supports2MbpsNonViterbi
RAIL_BLE_Supports2MbpsViterbi
RAIL_BLE_Supports2Mbps
RAIL_BLE_SupportsAntennaSwitching
RAIL_BLE_SupportsCodedPhy
RAIL_BLE_SupportsCte
RAIL_BLE_SupportsIQSampling
RAIL_BLE_SupportsPhySwitchToRx
RAIL_BLE_SupportsQuuppa
RAIL_BLE_SupportsSignalIdentifier
RAIL_BLE_SupportsSimulscanPhy
RAIL_SupportsProtocolIEEE802154
RAIL_IEEE802154_SupportsCoexPhy
RAIL_SupportsIEEE802154Band2P4

RAIL_SupportsThermalProtection
RAIL_IEEE802154_SupportsRxChannelSwitching
RAIL_IEEE802154_SupportsCustom1Phy
RAIL_IEEE802154_SupportsFemPhy
RAIL_IEEE802154_SupportsCancelFramePendingLookup
RAIL_IEEE802154_SupportsEarlyFramePendingLookup
RAIL_IEEE802154_SupportsDualPaConfig
RAIL_IEEE802154_SupportsEEEnhancedAck
RAIL_IEEE802154_SupportsEMultipurposeFrames
RAIL_IEEE802154_SupportsESubsetGB868
RAIL_IEEE802154_SupportsG4ByteCrc
RAIL_IEEE802154_SupportsGDynFec
RAIL_SupportsProtocolWiSUN
RAIL_IEEE802154_SupportsGModeSwitch
RAIL_IEEE802154_SupportsGSubsetGB868
RAIL_IEEE802154_SupportsGUnwhitenedRx
RAIL_IEEE802154_SupportsGUnwhitenedTx
RAIL_WMBUS_SupportsSimultaneousTCRx
RAIL_SupportsProtocolZWave
RAIL_ZWAVE_SupportsConcPhy
RAIL_ZWAVE_SupportsEnergyDetectPhy
RAIL_ZWAVE_SupportsRegionPti
RAIL_IEEE802154_SupportsSignalIdentifier
RAIL_SupportsFastRx2Rx
RAIL_SupportsCollisionDetection
RAIL_SupportsProtocolSidewalk
RAIL_SupportsTrustZoneSecurePeripherals
RAIL_SupportsPrsLnaBypass
RAIL_SupportsProtocolBTC
Front End Module Voltage Detection
RAIL_VdetConfig_t
 mode
 resolution
 delayUs
RAIL_Vdet_Mode_t
RAIL_Vdet_Resolution_t
RAIL_Vdet_Status_t
RAIL_ConfigVdet
RAIL_GetVdetConfig
RAIL_EnableVdet
RAIL_IsVdetEnabled
RAIL_GetVdet
General
RAIL_Version_t

- hash
- major
- minor
- rev
- build
- flags
- multiprotocol
- RAILSched_Config_t
 - buffer
- RAIL_Config_t
 - eventsCallback
 - protocol
 - scheduler
 - buffer
- EFR32xG2x
- Slxx3xx
- RAIL_Handle_t
- RAIL_Status_t
- RAIL_InitCompleteCallbackPtr_t
- RAIL_StateBuffer_t
- RAIL_GetVersion
- RAIL_AddStateBuffer3
- RAIL_AddStateBuffer4
- RAIL_UseDma
- RAIL_CopyDeviceInfo
- RAIL_Init
- RAIL_IsInitialized
- RAIL_GetRadioEntropy
- Multiprotocol
 - RAIL_SchedulerInfo_t
 - priority
 - slipTime
 - transactionTime
- EFR32xG2x
- Slxx3xx
- RAIL_SchedulerStatus_t
- RAIL_TaskType_t
- RAIL_YieldRadio
- RAIL_GetSchedulerStatus
- RAIL_GetSchedulerStatusAlt
- RAIL_SetTaskPriority
- RAIL_GetTransitionTime
- RAIL_SetTransitionTime
- PTI Packet Trace

RAIL_PtiConfig_t

mode

baud

doutLoc

doutPort

doutPin

dclkLoc

dclkPort

dclkPin

dframeLoc

dframePort

dframePin

RAIL_PtiMode_t

RAIL_PtiProtocol_t

RAIL_ConfigPti

RAIL_GetPtiConfig

RAIL_EnablePti

RAIL_SetPtiProtocol

RAIL_GetPtiProtocol

Protocol-specific

BLE

RAIL_BLE_State_t

crclnit

accessAddress

channel

disableWhitening

whitelnit

Angle of Arrival/Departure

RAIL_BLE_AoxConfig_t

aoxOptions

cteBuffSize

cteBuffAddr

antArrayAddr

antArraySize

RAIL_BLE_AoxAntennaPortPins_t

antPort

antPin

RAIL_BLE_AoxAntennaConfig_t

antPortPin

antCount

RAIL_BLE_AoxOptions_t

RAIL_BLE_LockCteBuffer

RAIL_BLE_CteBufferIsLocked

RAIL_BLE_GetCteSampleOffset

- RAIL_BLE_GetCteSampleRate
- RAIL_BLE_ConfigAox
- RAIL_BLE_InitCte
- RAIL_BLE_ConfigAoxAntenna
- BLE Radio Configurations
 - RAIL_BLE_Phy1Mbps
 - RAIL_BLE_Phy2Mbps
 - RAIL_BLE_Phy1MbpsViterbi
 - RAIL_BLE_Phy2MbpsViterbi
 - RAIL_BLE_Phy2MbpsAox
 - RAIL_BLE_Phy125kbps
 - RAIL_BLE_Phy500kbps
 - RAIL_BLE_PhySimulscan
 - RAIL_BLE_PhyQuuppa
- BLE TX Channel Hopping
 - RAIL_BLE_TxChannelHoppingConfigEntry_t
 - delay
 - phy
 - logicalChannel
 - railChannel
 - disableWhitening
 - crclnit
 - accessAddress
 - RAIL_BLE_TxChannelHoppingConfig_t
 - buffer
 - bufferLength
 - numberOfChannels
 - reserved
 - entries
 - RAIL_BLE_TxRepeatConfig_t
 - iterations
 - repeatOptions
 - delay
 - channelHopping
 - delayOrHop
 - RAIL_BLE_SetNextTxRepeat
- RAIL_BLE_Coding_t
- RAIL_BLE_Phy_t
- RAIL_BLE_SignalIdentifierMode_t
- RAIL_BLE_Init
- RAIL_BLE_Deinit
- RAIL_BLE_IsEnabled
- RAIL_BLE_ConfigPhyQuuppa
- RAIL_BLE_ConfigPhy1MbpsViterbi

RAIL_BLE_ConfigPhy1Mbps
RAIL_BLE_ConfigPhy2MbpsViterbi
RAIL_BLE_ConfigPhy2Mbps
RAIL_BLE_ConfigPhyCoded
RAIL_BLE_ConfigPhySimulScan
RAIL_BLE_ConfigChannelRadioParams
RAIL_BLE_PhySwitchToRx
RAIL_BLE_ConfigSignalIdentifier
RAIL_BLE_EnableSignalDetection

IEEE 802.15.4

RAIL_IEEE802154_Address_t
 shortAddress
 longAddress
 @4
 length
 filterMask

RAIL_IEEE802154_AddrConfig_t
 panId
 shortAddr
 longAddr

RAIL_IEEE802154_Config_t
 addresses
 ackConfig
 timings
 framesMask
 promiscuousMode
 isPanCoordinator
 defaultFramePendingInOutgoingAcks

RAIL_IEEE802154_RxChannelSwitchingCfg_t
 buffer
 bufferBytes
 channels

RAIL_IEEE802154_ModeSwitchPhr_t
 phyModelId
 phr

IEEE 802.15.4 Radio Configurations

RAIL_IEEE802154_Phy2p4GHz
RAIL_IEEE802154_Phy2p4GHzAntDiv
RAIL_IEEE802154_Phy2p4GHzCoex
RAIL_IEEE802154_Phy2p4GHzAntDivCoex
RAIL_IEEE802154_Phy2p4GHzFem
RAIL_IEEE802154_Phy2p4GHzAntDivFem
RAIL_IEEE802154_Phy2p4GHzCoexFem
RAIL_IEEE802154_Phy2p4GHzAntDivCoexFem

RAIL_IEEE802154_Phy2p4GHzCustom1
RAIL_IEEE802154_PhyGB863MHz
RAIL_IEEE802154_PhyGB915MHz
RAIL_IEEE802154_Phy2p4GHzRxChSwitching
RAIL_IEEE802154_AddressLength_t
RAIL_IEEE802154_PtiRadioConfig_t
RAIL_IEEE802154_EOptions_t
RAIL_IEEE802154_GOptions_t
RAIL_IEEE802154_CcaMode_t
RAIL_IEEE802154_SignalIdentifierMode_t
RAIL_IEEE802154_Init
RAIL_IEEE802154_Config2p4GHzRadio
RAIL_IEEE802154_Config2p4GHzRadioAntDiv
RAIL_IEEE802154_Config2p4GHzRadioAntDivCoex
RAIL_IEEE802154_Config2p4GHzRadioCoex
RAIL_IEEE802154_Config2p4GHzRadioFem
RAIL_IEEE802154_Config2p4GHzRadioAntDivFem
RAIL_IEEE802154_Config2p4GHzRadioCoexFem
RAIL_IEEE802154_Config2p4GHzRadioAntDivCoexFem
RAIL_IEEE802154_Config2p4GHzRadioCustom1
RAIL_IEEE802154_ConfigGB863MHzRadio
RAIL_IEEE802154_ConfigGB915MHzRadio
RAIL_IEEE802154_Deinit
RAIL_IEEE802154_IsEnabled
RAIL_IEEE802154_GetPtiRadioConfig
RAIL_IEEE802154_SetAddresses
RAIL_IEEE802154_SetPanId
RAIL_IEEE802154_SetShortAddress
RAIL_IEEE802154_SetLongAddress
RAIL_IEEE802154_SetPanCoordinator
RAIL_IEEE802154_SetPromiscuousMode
RAIL_IEEE802154_ConfigEOptions
RAIL_IEEE802154_ConfigGOptions
RAIL_IEEE802154_ComputeChannelFromPhyModelId
RAILCb_IEEE802154_IsModeSwitchNewChannelValid
RAIL_IEEE802154_AcceptFrames
RAIL_IEEE802154_EnableEarlyFramePending
RAIL_IEEE802154_EnableDataFramePending
RAIL_IEEE802154_SetFramePending
RAIL_IEEE802154_GetAddress
RAIL_IEEE802154_WriteEnhAck
RAIL_IEEE802154_SetRxToEnhAckTx
RAIL_IEEE802154_ConvertRssiToLqi

RAIL_IEEE802154_ConvertRssiToEd
RAIL_IEEE802154_ConfigSignalIdentifier
RAIL_IEEE802154_EnableSignalDetection
RAIL_IEEE802154_ConfigCcaMode
RAIL_IEEE802154_ConfigRxChannelSwitching

Multi-Level Frequency Modulation

RAIL_MFM_PingPongBufferConfig_t
 pBuffer0
 pBuffer1
 bufferSizeWords

RAIL_SetMfmPingPongFifo

Sidewalk Radio Configurations

RAIL_Sidewalk_Phy2GFSK50kbps
RAIL_Sidewalk_ConfigPhy2GFSK50kbps

Wireless M-Bus

RAIL_WMBUS_Phy_t
RAIL_WMBUS_Config

Z-Wave

RAIL_ZWAVE_Config_t
 options
 ackConfig
 timings

RAIL_ZWAVE_LrAckData_t
 noiseFloorDbm
 txPowerDbm
 receiveRssiDbm

RAIL_ZWAVE_BeamRxConfig_t
 channelHoppingConfig
 receiveConfig_100
 receiveConfig_40

RAIL_ZWAVE_RegionConfig_t
 frequency
 maxPower
 baudRate
 regionId
 regionSpecific

RAIL_ZWAVE_IrcalVal_t
 imageRejection

RAIL_ZWAVE_Options_t

RAIL_ZWAVE_NodeId_t

RAIL_ZWAVE_Homeld_t

RAIL_ZWAVE_HomeldHash_t

RAIL_ZWAVE_Baud_t

RAIL_ZWAVE_RegionId_t

RAIL_RxChannelHoppingParameters_t
RAIL_ZWAVE_REGION_EU
RAIL_ZWAVE_REGION_US
RAIL_ZWAVE_REGION_ANZ
RAIL_ZWAVE_REGION_HK
RAIL_ZWAVE_REGION_MY
RAIL_ZWAVE_REGION_IN
RAIL_ZWAVE_REGION_JP
RAIL_ZWAVE_REGION_JPED
RAIL_ZWAVE_REGION_RU
RAIL_ZWAVE_REGION_IL
RAIL_ZWAVE_REGION_KR
RAIL_ZWAVE_REGION_KRED
RAIL_ZWAVE_REGION_CN
RAIL_ZWAVE_REGION_US_LR1
RAIL_ZWAVE_REGION_US_LR2
RAIL_ZWAVE_REGION_US_LR3
RAIL_ZWAVE_REGION_EU_LR1
RAIL_ZWAVE_REGION_EU_LR2
RAIL_ZWAVE_REGION_EU_LR3
RAIL_ZWAVE_REGION_INVALID
RAIL_ZWAVE_ConfigRegion
RAIL_ZWAVE_PerformIrcal
RAIL_ZWAVE_Init
RAIL_ZWAVE_Deinit
RAIL_ZWAVE_IsEnabled
RAIL_ZWAVE_ConfigOptions
RAIL_ZWAVE_SetNodeId
RAIL_ZWAVE_SetHomeId
RAIL_ZWAVE_GetBeamNodeId
RAIL_ZWAVE_GetBeamHomeIdHash
RAIL_ZWAVE_GetBeamChannelIndex
RAIL_ZWAVE_GetLrBeamTxPower
RAIL_ZWAVE_GetBeamRssi
RAIL_ZWAVE_SetTxLowPower
RAIL_ZWAVE_GetTxLowPower
RAIL_ZWAVE_SetTxLowPowerDbm
RAIL_ZWAVE_GetTxLowPowerDbm
RAIL_ZWAVE_ReceiveBeam
RAIL_ZWAVE_ConfigBeamRx
RAIL_ZWAVE_SetDefaultRxBeamConfig
RAIL_ZWAVE_GetRxBeamConfig
RAIL_ZWAVE_ConfigRxChannelHopping

RAIL_ZWAVE_GetRegion

RAIL_ZWAVE_SetLrAckData

RF Sense

RAIL_RfSenseSelectiveOokConfig_t

band

syncWordNumBytes

syncWord

cb

RAIL_RfSenseBand_t

RAIL_RfSense_CallbackPtr_t

RAIL_StartRfSense

RAIL_StartSelectiveOokRfSense

RAIL_ConfigRfSenseSelectiveOokWakeupPhy

RAIL_SetRfSenseSelectiveOokWakeupPayload

RAIL_IsRfSensed

RX Channel Hopping

RAIL_RxChannelHoppingConfigMultiMode_t

syncDetect

preambleSense

timingSense

timingReSense

status

RAIL_RxChannelHoppingConfigEntry_t

channel

mode

parameter

delay

delayMode

options

rssThresholdDbm

reserved2

RAIL_RxChannelHoppingConfig_t

buffer

bufferLength

numberOfChannels

entries

RAIL_RxDutyCycleConfig_t

mode

parameter

delay

delayMode

options

rssThresholdDbm

reserved2

EFR32xG2x
Slxx3xx
RAIL_RxChannelHoppingMode_t
RAIL_RxChannelHoppingDelayMode_t
RAIL_RxChannelHoppingOptions_t
RAIL_RxChannelHoppingParameter_t
RAIL_ConfigRxChannelHopping
RAIL_EnableRxChannelHopping
RAIL_TriggerRxChannelHop
RAIL_GetChannelHoppingRssi
RAIL_ConfigRxDutyCycle
RAIL_EnableRxDutyCycle
RAIL_GetDefaultRxDutyCycleConfig

Radio Configuration

RAIL_FrameType_t
 frameLen
 offset
 mask
 isValid
 addressFilter
RAIL_AlternatePhy_t
 baseFrequency
 channelSpacing
 numberOfChannels
 minIf_kHz
 minBaself_kHz
 isOfdmModem
 rateInfo
 hwModemAgcCtrl1

RAIL_ChannelConfigEntry_t
 phyConfigDeltaAdd
 baseFrequency
 channelSpacing
 physicalChannelOffset
 channelNumberStart
 channelNumberEnd
 maxPower
 attr
 entryType
 reserved
 stackInfo
 alternatePhy
RAIL_ChannelConfig_t
 phyConfigBase

phyConfigDeltaSubtract
configs
length
signature
xtalFrequencyHz
RAIL_ChannelMetadata_t
channel
reserved
frequency
RAIL_StackInfoCommon_t
protocolId
phyId
RAIL_ChannelConfigEntryAttr_t
RAIL_ChannelConfigEntryType_t
RAIL_RadioConfig_t
RAIL_RadioConfigChangedCallback_t
RAIL_SetFixedLength
RAIL_ConfigChannels
RAIL_ConfigChannelsAlt
RAIL_GetChannelMetadata
RAIL_IsValidChannel
RAIL_PrepareChannel
RAIL_GetChannel
RAIL_GetChannelAlt
RAIL_GetSymbolRate
RAIL_GetBitRate
RAIL_SetPaCTune
RAIL_GetSyncWords
RAIL_ConfigSyncWords
RAIL_GetWhiteningInitVal
RAIL_GetCrcInitVal
RAIL_SetWhiteningInitVal
RAIL_SetCrcInitVal
RAIL_ResetWhiteningInitVal
RAIL_ResetCrcInitVal
Receive
RAIL_ScheduleRxConfig_t
start
startMode
end
endMode
rxTransitionEndSchedule
hardWindowEnd
RAIL_RxPacketInfo_t

packetStatus
packetBytes
firstPortionBytes
firstPortionData
lastPortionData
filterMask

RAIL_RxPacketDetails_t

timeReceived
crcPassed
isAck
rssi
lqi
syncWordId
subPhyId
antennaId
channelHoppingChannelIndex
channel

RAIL_PrsLnaBypassConfig_t

timeoutUs
threshold
deltaRssiDbm
prsChannel
polarity

Address Filtering

RAIL_AddrConfig_t

offsets
sizes
matchTable

RAIL_AddrFilterMask_t

RAIL_ConfigAddressFilter
RAIL_EnableAddressFilter
RAIL_IsAddressFilterEnabled
RAIL_ResetAddressFilter
RAIL_SetAddressFilterAddress
RAIL_SetAddressFilterAddressMask
RAIL_EnableAddressFilterAddress

Packet Information

RAIL_GetRxPacketInfo
RAIL_GetRxIncomingPacketInfo
RAIL_CopyRxPacket
RAIL_GetRxPacketDetails
RAIL_GetRxPacketDetailsAlt
RAIL_GetRxTimePreambleStart
RAIL_GetRxTimePreambleStartAlt

RAIL_GetRxTimeSyncWordEnd
RAIL_GetRxTimeSyncWordEndAlt
RAIL_GetRxTimeFrameEnd
RAIL_GetRxTimeFrameEndAlt

RAIL_RxOptions_t
RAIL_RxPacketStatus_t
RAIL_RxPacketHandle_t
RAIL_ConvertLqiCallback_t
RAIL_ConfigRxOptions
RAIL_IncludeFrameTypeLength
RAILCb_ConfigFrameTypeLength
RAIL_StartRx
RAIL_ScheduleRx
RAIL_EnablePrsLnaBypass
RAIL_HoldRxPacket
RAIL_PeekRxPacket
RAIL_ReleaseRxPacket
RAIL_GetRssi
RAIL_GetRssiAlt
RAIL_StartAverageRssi
RAIL_IsAverageRssiReady
RAIL_GetAverageRssi
RAIL_SetRssiOffset
RAIL_GetRssiOffset
RAIL_SetRssiDetectThreshold
RAIL_GetRssiDetectThreshold
RAIL_GetRxIncomingPacketRssi
RAIL_ConvertLqi

Retiming

RAIL_RetimeOptions_t
RAIL_ConfigRetimeOptions
RAIL_GetRetimeOptions
RAIL_ChangedDcdc

Sleep

RAIL_TimerSyncConfig_t
 prsChannel
 rtccChannel
 sleep
EFR32xG2x
Slxx3xx
RAIL_SleepConfig_t
RAILCb_ConfigSleepTimerSync
RAIL_ConfigSleep
RAIL_ConfigSleepAlt

RAIL_Sleep
RAIL_Wake
RAIL_InitPowerManager
RAIL_DeinitPowerManager

State Transitions

RAIL_StateTiming_t
idleToRx
txToRx
idleToTx
rxToTx
rxSearchTimeout
txToRxSearchTimeout
txToTx

RAIL_StateTransitions_t
success
error

EFR32xG2x

Slxx3xx

RAIL_RadioState_t

RAIL_RadioStateDetail_t

RAIL_IdleMode_t

RAIL_TransitionTime_t

RAIL_SetRxTransitions

RAIL_GetRxTransitions

RAIL_SetTxTransitions

RAIL_GetTxTransitions

RAIL_SetNextTxRepeat

RAIL_GetTxPacketsRemaining

RAIL_SetStateTiming

RAIL_Idle

RAIL_GetRadioState

RAIL_GetRadioStateDetail

RAIL_EnableCacheSynthCal

System Timing

RAIL_MultiTimer_t
absOffset
relPeriodic
callback
cbArg
next
priority
isRunning
doCallback

RAIL_PacketTimeStamp_t

- packetTime
- totalPacketBytes
- timePosition
- packetDurationUs
- RAIL_MultiTimer
- RAIL_TimeMode_t
- RAIL_PacketTimePosition_t
- RAIL_Time_t
- RAIL_TimerCallback_t
- RAIL_MultiTimerCallback_t
- RAIL_GetTime
- RAIL_SetTime
- RAIL_DelayUs
- RAIL_SetTimer
- RAIL_GetTimer
- RAIL_CancelTimer
- RAIL_IsTimerExpired
- RAIL_IsTimerRunning
- RAIL_ConfigMultiTimer
- RAIL_SetMultiTimer
- RAIL_CancelMultiTimer
- RAIL_IsMultiTimerRunning
- RAIL_IsMultiTimerExpired
- RAIL_GetMultiTimer

TX Channel Hopping

- RAIL_TxChannelHoppingConfigEntry_t
 - channel
 - reserved
 - delay
- RAIL_TxChannelHoppingConfig_t
 - buffer
 - bufferLength
 - numberOfChannels
 - reserved
 - entries

Thermal Protection

- RAIL_ChipTempConfig_t
 - enable
 - coolDownK
 - thresholdK
- RAIL_ChipTempMetrics_t
 - tempK
 - minTempK
 - maxTempK

- resetPending
- reservedChipTemp
- RAIL_ConfigThermalProtection
- RAIL_GetThermalProtection
- RAIL_GetTemperature
- Transmit
 - RAIL_TxPacketDetails_t
 - timeSent
 - isAck
 - RAIL_ScheduleTxConfig_t
 - when
 - mode
 - txDuringRx
 - RAIL_CsmaConfig_t
 - csmaMinBoExp
 - csmaMaxBoExp
 - csmaTries
 - ccaThreshold
 - ccaBackoff
 - ccaDuration
 - csmaTimeout
 - RAIL_LbtConfig_t
 - lbtMinBoRand
 - lbtMaxBoRand
 - lbtTries
 - lbtThreshold
 - lbtBackoff
 - lbtDuration
 - lbtTimeout
 - RAIL_SyncWordConfig_t
 - syncWordBits
 - syncWord1
 - syncWord2
 - RAIL_TxRepeatConfig_t
 - iterations
 - repeatOptions
 - delay
 - channelHopping
 - delayOrHop
- Packet Transmit
 - RAIL_StartTx
 - RAIL_StartScheduledTx
 - RAIL_StartCcaCsmaTx
 - RAIL_StartCcaLbtTx

RAIL_StartScheduledCcaCsmaTx

RAIL_StartScheduledCcaLbtTx

Power Amplifier (PA)

RAIL_TxPowerSettingEntry_t

paPowerSetting

minPaPowerDdbm

maxPaPowerDdbm

currentPaPowerDdbm

RAIL_TxPowerConfig_t

mode

voltage

rampTime

RAIL_PaAutoModeConfigEntry_t

min

max

mode

band

sl_rail_nvm_pa_curve_t

curve_min_ddbm

curve_max_ddbm

curve_segments

sl_rail_nvm_pa_table_t

ddbm_values

sl_rail_nvm_pa_descriptor_t

algorithm

num_segments_or_entries

min

max

sl_rail_nvm_pa_config_t

version

num_descriptors

pa_voltage

signature

pa_descriptors

curve

table

pa_curve_or_table

EFR32xG2x

PA Curve Conversions

RAIL_TxPowerCurveSegment_t

maxPowerLevel

slope

intercept

RAIL_TxPowerCurves_t

- maxPower
- minPower
- powerParams
- RAIL_TxPowerCurvesConfig_t
 - txPower24HpCurves
 - txPowerSgCurves
 - txPower24LpCurves
 - piecewiseSegments
- RAIL_TxPowerCurveAlt_t
 - maxPower
 - minPower
 - powerParams
- RAIL_PowerConversion_t
 - powerCurve
 - mappingTable
- RAIL_PaDescriptor_t
 - algorithm
 - segments
 - min
 - max
 - conversion
- RAIL_TxPowerCurvesConfigAlt_t
 - curves
 - signature
 - paVoltage
- RAIL_PaConversionAlgorithm_t
- RAIL_TxPowerCurvesVbat
- RAIL_TxPowerCurvesDcdc
- RAIL_InitTxPowerCurves
- RAIL_InitTxPowerCurvesAlt
- RAIL_GetTxPowerCurve
- RAIL_GetTxPowerCurveLimits
- sl_rail_util_pa_init
- sl_rail_util_pa_get_tx_power_config_2p4ghz
- sl_rail_util_pa_get_tx_power_config_subghz
- sl_rail_util_pa_get_tx_power_config_ofdm
- sl_rail_util_pa_on_channel_config_change
- Slxx3xx
- RAIL_TxPowerMode_t
- RAIL_PaBand_t
- RAIL_TxPower_t
- RAIL_TxPowerLevel_t
- RAIL_PaPowerSetting_t
- RAIL_PaAutoModeConfig

RAIL_ConfigTxPower
RAIL_GetTxPowerConfig
RAIL_SetTxPower
RAIL_GetTxPower
RAIL_ConvertRawToDbm
RAIL_ConvertDbmToRaw
RAIL_ConvertDbmToPowerSettingEntry
RAIL_VerifyTxPowerCurves
RAIL_SetTxPowerDbm
RAIL_GetTxPowerDbm
RAIL_GetPowerSettingTable
RAIL_SetPaPowerSetting
RAIL_GetPaPowerSetting
RAIL_EnablePaAutoMode
RAIL_IsPaAutoModeEnabled
RAILCb_PaAutoModeDecision
RAIL_ConfigPaAutoEntry
sl_rail_util_pa_nvm_write_mode
sl_rail_util_pa_nvm_read_mode
sl_rail_util_pa_nvm_write_config
sl_rail_util_pa_nvm_read_config
RAIL_StopMode_t
RAIL_TxOptions_t
RAIL_ScheduledTxDuringRx_t
RAIL_TxRepeatOptions_t
RAIL_StopTx
RAIL_SetCcaThreshold
RAIL_GetTxPacketDetails
RAIL_GetTxPacketDetailsAlt
RAIL_GetTxPacketDetailsAlt2
RAIL_GetTxTimePreambleStart
RAIL_GetTxTimePreambleStartAlt
RAIL_GetTxTimeSyncWordEnd
RAIL_GetTxTimeSyncWordEndAlt
RAIL_GetTxTimeFrameEnd
RAIL_GetTxTimeFrameEndAlt
RAIL_EnableTxHoldOff
RAIL_IsTxHoldOffEnabled
RAIL_SetTxAltPreambleLength
TrustZone
RAIL_TZ_Config_t
 changedDcdcCallback
 configAntennaGpioCallback
 radioClockEnableCallback

getRadioClockFreqHzCallback
rfecaClockEnableCallback
rfecalsClockEnabledCallback
readInternalTemperatureCallback
enableSecureRadiolrqsCallback
disableSecureRadiolrqsCallback
radioPerformM2mLdmaCallback
configureHfxoCallback
isCmuSecure
isEmuSecure
isGpioSecure
isLdmaSecure
isHfxoSecure
isPrsSecure
isSysrtcSecure
RAIL_TZ_ChangedDcdcCallbackPtr_t
RAIL_TZ_ConfigAntennaGpioCallbackPtr_t
RAIL_TZ_RadioClockEnableCallbackPtr_t
RAIL_TZ_GetRadioClockFreqHzCallbackPtr_t
RAIL_TZ_RfecaClockEnableCallbackPtr_t
RAIL_TZ_RfecalsClockEnabledCallbackPtr_t
RAIL_TZ_ReadInternalTemperatureCallbackPtr_t
RAIL_TZ_EnableSecureRadiolrqsCallbackPtr_t
RAIL_TZ_DisableSecureRadiolrqsCallbackPtr_t
RAIL_TZ_RadioPerformM2mLdmaCallbackPtr_t
RAIL_TZ_ConfigureHfxoCallbackPtr_t
RAIL_TZ_InitNonSecure
RAIL_TZ_InitSecure
RAIL_TZ_CheckPeripheralsSecureStates
RAIL_TZ_RadioClockEnable
RAIL_TZ_RfecaClockEnable
RAIL_TZ_RfecalsClockEnabled
RAIL_TZ_ReadInternalTemperature
RAIL_TZ_EnableSecureRadiolrqs
RAIL_TZ_DisableSecureRadiolrqs
RAIL_TZ_RadioPerformM2mLdma
RAIL_TZ_ConfigureHfxo
RAIL_TZ_ConfigAntennaGpio

Components

RAIL Library
Angle of Arrival/Departure (AoX) Utility
Antenna Diversity Utility
BLE Channel Sounding (CS) Utility
BLE Channel Sounding (CS) Antenna Offset Utility

- Built-in PHYs for alternate HFXO frequencies
- Callbacks Utility
- Coexistence Utility
- Direct Memory Access (DMA) Utility
- Front End Module Voltage Detection (VDET) Utility
- Front End Module (FEM) Utility
- Initialization Utility
- Packet Trace Interface (PTI) Utility
- Power Amplifier (PA) Utility
- Protocol Utility
- Recommended Utility
- RF Path Utility
- Received Signal Strength Indicator (RSSI) Utility
- RAILtest Application Core
- RAILtest Application Graphics
- Thermistor Utility
- RF Path Switch Utility
- Radio Sequencer Image Selection Utility
- Software Modem (SFM) sequencer image selection Utility

EFR32

- RAIL Multiprotocol
- Device Information Utility
- PA Power Configs in persistent memory
- Release Notes
 - RAIL Changelist
 - RAILtest Changelist
 - API Changes
- Multi-Protocol Transition Time Utility
- Deprecated List
- RAIL Library

Examples

- About RAIL Examples
- RAIL SDK Services

- Extension
 - Simple OS
 - app_task_init
 - app_task_notify
 - print_sample_app_name

Utility

- Channel Selector
 - Configurations
 - set_configured_channel
 - set_selected_channel
 - get_selected_channel

restart_rx_channel

PHY Selector

set_selected_phy

get_selected_phy

Packet Assistant

Configurations

Type definitions

RAIL_SDK_Protocol_t

validation_check

update_assistant_pointers

unpack_packet

prepare_packet

get_print_packet_info

set_print_packet_info

printf_rx_packet

get_wisun_ofdm_rate

set_wisun_ofdm_rate

get_wisun_ofdm_scrambler

set_wisun_ofdm_scrambler

get_wisun_fsk_fcs

set_wisun_fsk_fcs

get_wisun_fsk_whitening

set_wisun_fsk_whitening

get_sun_oqpsk_spreading_mode

set_sun_oqpsk_spreading_mode

get_sun_oqpsk_rate_mode

set_sun_oqpsk_rate_mode

get_sidewalk_fcs_type

set_sidewalk_fcs_type

get_sidewalk_whitening

set_sidewalk_whitening

Power Manager Integration

sl_rail_sdk_sleep_init

Wireless M-Bus

Wireless M-Bus Button Pulse Counter

Configurations

sl_rail_sdk_wmbus_sensor_pulse_counter_init

sl_rail_sdk_wmbus_sensor_pulse_counter_deinit

sl_rail_sdk_wmbus_sensor_pulse_counter_measure

sl_rail_sdk_wmbus_sensor_pulse_counter_get_data

sl_rail_sdk_wmbus_sensor_pulse_counter_print

sl_rail_sdk_wmbus_sensor_pulse_counter_button_pressed

Wireless M-Bus Sensor Core

Configurations

Type definitions

`sl_rail_sdk_wmbus_sensor_t``id``name``init_fnc_ptr``deinit_fnc_ptr``measure_fnc_ptr``get_data_fnc_ptr``print_fnc_ptr``button_pressed_fnc_ptr``sl_rail_sdk_wmbus_sensor_init_t``sl_rail_sdk_wmbus_sensor_deinit_t``sl_rail_sdk_wmbus_sensor_measure_t``sl_rail_sdk_wmbus_sensor_get_data_t``sl_rail_sdk_wmbus_sensor_print_t``sl_rail_sdk_wmbus_sensor_button_pressed_t``sl_rail_sdk_wmbus_sensor_core_get_active_sensor``sl_rail_sdk_wmbus_sensor_core_set_active_sensor``sl_rail_sdk_wmbus_sensor_core_list_available_sensors``sl_rail_sdk_wmbus_sensor_core_get_sensor_count``sl_rail_sdk_wmbus_sensor_core_init``sl_rail_sdk_wmbus_sensor_core_lcd_init``sl_rail_sdk_wmbus_sensor_core_process`

Wireless M-Bus Support

Configurations

Type definitions

`sl_rail_sdk_wmbus_dll_header_t``IField``raw``function_code``fcv_dcf``fcv_acd``primary``detailed``c_field``manufacturer``id``version``device_type``detailed``address``sl_rail_sdk_wmbus_conf_word_t``raw``hop_counter`

- repeated_access
- content_of_message
- num_of_enc_blocks
- mode
- synchronized
- accessibility
- mode_0_5
- sl_rail_sdk_wmbus_stl_header_t
 - ci_field
 - access_number
 - status
 - conf_word
- sl_rail_sdk_wmbus_ltl_header_t
 - ci_field
 - raw
 - id
 - manufacturer
 - version
 - device_type
 - detailed
 - address
 - access_number
 - status
 - conf_word
- sl_rail_sdk_wmbus_sensor_data_t
 - vif
 - data
 - next
- sl_rail_sdk_wmbus_function_code_t
- sl_rail_sdk_wmbus_device_type_t
- sl_rail_sdk_wmbus_ci_field_t
- sl_rail_sdk_wmbus_status_t
- sl_rail_sdk_wmbus_accessibility_t
- sl_rail_sdk_wmbus_frame_dib_function_t
- sl_rail_sdk_wmbus_mode_t
- sl_rail_sdk_wmbus_phy_software
- sl_rail_sdk_wmbus_get_accessibility
- sl_rail_sdk_wmbus_set_accessibility
- sl_rail_sdk_wmbus_init
- sl_rail_sdk_wmbus_set_mode
- sl_rail_sdk_wmbus_get_selected_mode
- sl_rail_sdk_wmbus_get_meter_limited_acc_rx_start
- sl_rail_sdk_wmbus_get_meter_limited_acc_rx_stop
- sl_rail_sdk_wmbus_frame_add_payload_int8

[sl_rail_sdk_wmbus_frame_add_payload_int16](#)
[sl_rail_sdk_wmbus_frame_add_payload_int32](#)
[sl_rail_sdk_wmbus_frame_add_idle_filler](#)
[sl_rail_sdk_wmbus_frame_chars_to_mfield](#)
[sl_rail_sdk_wmbus_frame_mfield_to_chars](#)
[sl_rail_sdk_wmbus_frame_crypto5_init](#)
[sl_rail_sdk_wmbus_frame_crypto5_set_key](#)
[sl_rail_sdk_wmbus_frame_crypto5_get_key](#)
[sl_rail_sdk_wmbus_frame_crypto5_encrypt](#)
[sl_rail_sdk_wmbus_frame_crypto5_decrypt](#)
[sl_rail_sdk_wmbus_setup_frame](#)

[Wireless M-Bus Thermometer Sensor](#)

[Configurations](#)

[sl_rail_sdk_wmbus_sensor_thermometer_init](#)
[sl_rail_sdk_wmbus_sensor_thermometer_deinit](#)
[sl_rail_sdk_wmbus_sensor_thermometer_measure](#)
[sl_rail_sdk_wmbus_sensor_thermometer_get_data](#)
[sl_rail_sdk_wmbus_sensor_thermometer_print](#)
[sl_rail_sdk_wmbus_sensor_thermometer_button_pressed](#)

[Wireless M-Bus Virtual Water Meter Sensor](#)

[Configurations](#)

[sl_rail_sdk_wmbus_sensor_virtual_water_meter_init](#)
[sl_rail_sdk_wmbus_sensor_virtual_water_meter_deinit](#)
[sl_rail_sdk_wmbus_sensor_virtual_water_meter_measure](#)
[sl_rail_sdk_wmbus_sensor_virtual_water_meter_get_data](#)
[sl_rail_sdk_wmbus_sensor_virtual_water_meter_print](#)
[sl_rail_sdk_wmbus_sensor_virtual_water_meter_button_pressed](#)

[Migration Guide](#)

[Services](#)

[Flex SDK v3.x Range Test Demo User's Guide \(PDF\)](#)

[RAILtest User's Guide \(PDF\)](#)

[Using RAIL for Wireless M-Bus Applications with EFR32](#)

[Overview](#)

[Limitations](#)

[Using the Radio Configurator](#)

[Wireless M-Bus Support Component](#)

[Example Application](#)

[Changing the Mode of the Example](#)

[Training](#)

[Overview](#)

[Getting Started](#)

[Introduction](#)

[Transmit](#)

[Event Handling](#)

[Downloading Messages](#)

[Scheduling and Timestamping](#)

Basics

[Debugging](#)

[Calibrations](#)

[Power Manager](#)

[Idle Modes](#)

Further Topics

[Sleep Timer Synchronization](#)

[Tx/Rx Options](#)

[FIFO Mode](#)

[Low Duty Cycle Theory](#)

[Thread Safety](#)

[RAIL Config Files](#)

Multiprotocol

[Introduction](#)

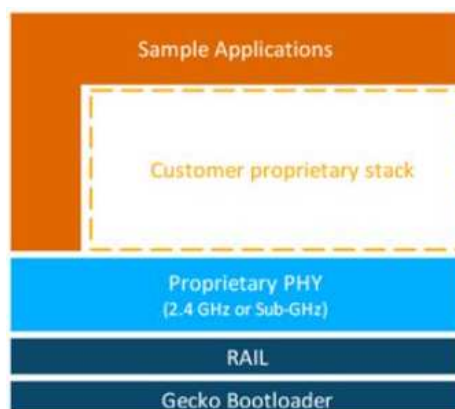
[MultiPHY Use Cases](#)

[DMP](#)

Developing with RAIL

Developing with Silicon Labs RAIL

Silicon Labs RAIL (Radio Abstraction Interface Layer) provides an intuitive and easily-customizable radio interface that is designed to support proprietary or standards-based wireless protocols. RAIL allows customers to adopt the latest RF technology without sacrificing their previous development investment. It also future-proofs the code migration to future EFR32 ICs. The unified radio software API abstracts the significant number of hardware registers and complexity of the lower-level radio block, allowing customers to focus on their proprietary wireless application development instead of mastering device-specific details. RAIL is delivered through the Proprietary Flex SDK component of the Gecko SDK Suite (GSDK).



Silicon Labs RAIL is the lowest layer for all networking stacks in Silicon Labs GSDK. RAIL supports a diverse set of radio configurations and functionality and is one of the key underlying technologies of Silicon Labs wireless products. The content on these pages is intended for developers who need to control their device's radio interface and performance through their embedded application.

For details about this release: Links to release notes are available on the [silabs.com Gecko SDK page](#).

For Silicon Labs' proprietary product information: See the [product pages on silabs.com](#).

For background about RAIL: [RAIL Fundamentals](#) is a good place to start.

To get started with development: See the [Getting Started section](#) to get started working with example applications.

If you are already in development: See the [Developer's Guide](#) or the [PHY Development section](#) for details or go directly to the [API Reference](#).

RAIL SDK

RAIL SDK Version 2.19.0 (June 16, 2025) - Release Notes

[Simplicity SDK Version 2025.6.0](#)

The RAIL SDK is a complete software development suite for proprietary wireless applications. It was previously part of the Proprietary SDK. Starting with the RAIL SDK 2.18.0 release, Proprietary SDK is split into RAIL SDK and Connect SDK.

Silicon Labs RAIL (Radio Abstraction Interface Layer) is an intuitive and easily-customizable radio interface layer designed to support both proprietary and standards-based wireless protocols.

The RAIL SDK is supplied with extensive documentation and sample applications.

Release Summary

[Key Features](#) | [API Changes](#) | [Bug Fixes](#) | [Chip Enablement](#)

Key Features

Upgraded all components and sample application to use RAIL 2.19.0.

API Changes

- All API that used RAIL 2.18.x typedef returns or params have been updated to use RAIL 2.19.0.
- `unpack_packet` got a new parameter `sl_rail_handle_t rail_handle`

Bug Fixes

- Fixed the Mode Switch sample application default channel start and end values to follow standard.
- Fixed issue that when sample application got the `receive 0` command it only stopped the CLI print of the packet not the receiving.

Chip Enablement

EFRxG29

Key Features

[New Features](#) | [Enhancements](#) | [Removed Features](#) | [Deprecated Features](#)

New Features

- `sl_main` support instead of deprecated `sl_system`, now `sl_main` will provide the `main.c` file in the root directory.
- RAIL 2.19.0 support, all APIs, variables, typedefs have been updated to follow the RAIL library changes. Some APIs have been changed to have newer versions with same functionality.

Enhancements

All sample application use `sl_main` instead of `sl_system`. `main.c` is now provided by the platform component.

Removed Features

None.

Deprecated Features

None.

API Changes

[New APIs](#) | [Modified APIs](#) | [Removed APIs](#) | [Deprecated APIs](#)

New APIs

None.

Modified APIs

Old API	Modified
uint16_t unpack_packet (uint8_t * rx_destination, const RAIL_RxPacketInfo_t * packet_information, uint8_t ** start_of_payload)	uint16_t unpack_packet (sl_rail_handle_t rail_handle, uint8_t *rx_destination, const sl_rail_rx_packet_info_t *packet_information, uint8_t **start_of_payload, phy_modulation_e modulation)

Removed APIs

None.

Deprecated APIs

None.

Bug Fixes

ID	Issue Description	GitHub / Salesforce Reference (if any)	Affected Software Variants, Hardware, Modes, Host Interfaces
1424608	Wrong default channel numbers in default radio configuration	None	• Mode Switch sample application • BRD2705A,BRD4270A,BRD4270B,BRD4271A,BRD4272A,BRD4276A,BRD4400A,BRD4400B,BRD4400C • SoC • Radio Configurator
1375765	CLI command `receive 0` only stopped the CLI print, not the receive.	None	• RAIL SDK sample applications • All boards • SoC • UART, EUART interface
1426032	Range Test remote Config did not work on boards without LCD support.	None	• Range Test sample apps • Thunderboards • SoC • UART, EUART interface

Chip Enablement

Chip Family	OPNs / Boards / OPN Combinations	Supported Software Variants (if applicable)	Supported Modes	Supported Host Interfaces
EFR32xG29	- OPN: EFR32BG29, EFR32MG29 - Boards: BRD4412A, BRD4413A, BRD4414A, BRD4420A	Standard	- SoC - NCP	SPI, EUSART, UART

Application Example Changes

[New Examples](#) | [Modified Examples](#) | [Removed Examples](#) | [Deprecated Examples](#)

New Examples

None.

Modified Examples

Example Name	Changes	Supported Software Variants if applicable	Supported Modes	Supported OPNs / Boards / OPN Combinations	Supported Host Interfaces
RAIL - SoC Energy Mod sample application	Updated to use the new EM mode	Standard	SoC	- OPN: All - Boards: All	All

Removed Examples

None.

Deprecated Examples

None.

Impact of Release Changes

[Impact Statements](#) | [Migration Guide](#)

Impact Statements

Change	Impact	Affected Software Variants if applicable	Affected Modes	Affected OPNs / Boards / OPN Combinations	Affected Host Interfaces
Update of the unpack_packet	Need to add sl_rail_handle new param to handle the new RAIL 2.19.0. APIs.	Standard	- SoC - NCP	- OPN: all - Boards: all	All

Migration Guide

Click [here](#) for the migration guide for deprecated, removed, and modified items.

Known Issues and Limitations

ID	Issue or Limitation Description	GitHub / Salesforce Reference (if any)	Workaround (if any)	Affected Software Variants, Hardware, Modes, Host Interfaces
1268208	The power consumption of DK2600 can't be optimized if configuration time for EM2 and if UART is being used, as the value of SL_IOSTREAM_USART_{instance}_BAUDRATE doesn't take effect. The default value used instead is 115200 bps. Typical use case is RAIL - SoC Wireless M-bus Meter with DK2600 board.	None	Update UART baud rate to 9600 bps at Simplicity Studio Admin Console. This is necessary for EM2.	- RAIL SDK sample applications - BRD2600A - SoC - UART interface
1274248	Range Test DMP instabilities in case of LTO.	None	Do not use LTO.	- Range Test DMP sample applications - All supported boards - SoC
1375450	Wi-SUN HAN phys aren't supported by Standard packet assembler and Packet Assistant components.	None	Do not use Wi-SUN HAN phys with RAIL SDK sample applications.	- RAIL SDK sample applications - All supported boards - SoC, NCP mode - UART, SPI interface
	Updating the Baremetal sample applications to RTOS causes build errors.	None	Follow the Platforms guide sl_main Baremetal to RTOS Application Transition	- RAIL SDK Baremetal sample applications - All supported boards - SoC, NCP mode - UART, SPI interface

Using This Release

[What's in the Release?](#) | [Compatible Software](#) | [Installation and Use](#) | [Help and Feedback](#)

What's in the Release?

RAIL SDK showcasing the usage of the RAIL library.

Compatible Software

None.

Installation and Use

To upgrade your existing software with this release, see instructions [here](#).

To run your first demo, see our [Getting Started Guides](#).

To kick start your development, see our [Developer's Guide](#).

For information about Secure Vault Integration, see [Secure Vault](#).

To review Security and Software Advisory notifications and manage your notification preferences:

1. Go to <https://community.silabs.com/>.
2. Log in with your account credentials.
3. Click your profile icon in the upper-right corner of the page.
4. Select **Notifications** from the dropdown menu.
5. In the Notifications section, go to the **My Product Notifications** tab to review historical Security and Software Advisory notifications
6. To manage your preferences, use the **Manage Notifications** tab to customize which product updates and advisories you receive.

To learn more about the software in this release, dive into our [online documentation](#).

Help and Feedback

- Contact [Silicon Labs Support](#).
- To use our **Ask AI** tool to get answers, see the search field at the top of [this page](#).

Note: Ask AI is experimental.

- Get help from our [developer community](#).

SDK Release and Maintenance Policy

See our [SDK Release and Maintenance Policy](#).

Overview

Getting Started with RAIL - Overview

To get started, download the Simplicity Studio® 5 (SSv5) Development environment as described in the [Simplicity Studio 5 User's Guide](#).

This will also prompt you to install the Gecko SDK (GSDK). The Radio Configurator tool and the RAIL library are installed with the GSDK as part of the Proprietary Flex package. The GSDK combines Silicon Labs wireless software development kits (SDKs) and Gecko Platform into a single, integrated package. The GSDK is your primary tool for developing in the Silicon Labs IoT Software ecosystem. All of Silicon Labs' stacks are written in-house to provide a seamless experience from silicon to tools, allowing you to unlock powerful features with ease, including:

- Abstraction of complex requirements like multiprotocol and pre-certification
- Industry-leading ability to support a large number of nodes
- Ultra-low power consumption
- Strong network reliability

Silicon Labs also helps future-proof your devices with over-the-air software and security updates, helping to minimize maintenance cost and improve your end user product experience.

Once you have downloaded Simplicity Studio and the GSDK, detailed instructions for using the RAIL examples and configuration tools are provided in the [Proprietary Flex Quick-Start Guide \(QSG168\)](#).

Information about the distinguishing features of different EFR32 families that are most relevant to porting proprietary wireless applications between them is provided in [EFR32 Migration Guide for Proprietary Applications](#). This is also helpful when selecting an initial target platform for proprietary wireless solutions.

Note: The recommended method to get started with the GSDK is to first install Simplicity Studio 5, which will set up your development environment and walk you through the GSDK installation. Alternatively, GSDK and other required tools may be installed manually from the [GitHub GSDK site](#).

Development Tools

As well as Simplicity Studio, Silicon Labs provides the following development tools.

Network Analyzer: SSv5's Network Analyzer enables debugging of complex wireless systems. This tool captures a trace of wireless network activity that can be examined in detail live or at a later time. See the [Network Analyzer section](#) of the Simplicity Studio 5 User's Guide for more information.

Wireshark: Is this of interest to RAIL? Wireshark is the recommended network protocol analyzer for use with **for example, Wi-SUN**. Download instructions are provided for [Windows/Mac users](#) or [Linux users](#). Simplicity Studio® 5 supports [live interaction](#) between the application running on a Silicon Labs device and Wireshark.

Energy Profiler: SSv5's Energy Profiler enables you to visualize the energy consumption of individual devices, multiple devices on one target system, or a network of interacting wireless devices to analyze and improve the power performance of these systems. Real-time information on current consumption is correlated with the program counter providing advanced energy software monitoring capabilities. It also provides a basic level of integration with the Network Analyzer network analysis tool. See the [Energy Profiler section](#) of the Simplicity Studio 5 User's Guide for more information.

Simplicity Commander: Simplicity Commander is a single, all-purpose tool to be used in a production environment. It is invoked using a simple Command Line Interface (CLI) that is also scriptable. Simplicity Commander enables customers to complete essential tasks such as configuring and building applications and bootloaders and flashing images to their devices. Simplicity Commander is available through SSv5 or can be downloaded through [system-specific installers](#). The [Simplicity Commander User's Guide](#) provides more information.

Silicon Labs Configurator (SLC): SLC offers command-line access to application configuration and generation functions. [Software Project Generation and Configuration with SLC-CLI](#) provides instructions on downloading and using the SLC-CLI

tool.

EFR32 Migration Guide for Proprietary Applications

EFR32 Migration Guide for Proprietary Applications

This section replaces *AN1244: EFR32 Migration Guide for Proprietary Applications*. Further updates to this application note will be provided here.

This application note presents a high-level collection of elements that differentiate Wireless MCU EFR32 generations from each other, as well as more detailed descriptions of features exclusive to the newest EFR32 generations. This information is designed to support those considering migrating proprietary applications from one EFR32 generation to another.

Proprietary is supported on all EFR32FG devices. For others, check the device's data sheet under Ordering Information > Protocol Stack to see if Proprietary is supported. In Proprietary SDK version 2.7.n, Connect is not supported on EFR32xG22.

Introduction

This application note uses the following terms to classify devices in accordance with the EFR32 device datasheets:

- Device Configuration: EFR32xG1, EFR32xGx1, EFR32xGx2, EFR32xGx3, EFR32xGx4, and EFR32xGx5 devices
- Series: EFR32xG1x (including EFR32xG1 devices) and EFR32xG2x devices
- Device Family: EFR32MGxx, EFR32BGxx, EFR32FGxx, and EFR32ZGxx devices

Device generations are defined as the combinations of the device configuration and series. The complete list of EFR32 device generations is: EFR32xG1, EFR32xG12, EFR32xG13, EFR32xG14, EFR32xG21, EFR32xG22, EFR32xG23, EFR32xG24, and EFR32xG25.

For more information on the existing combinations of EFR32 generations and families, visit [silabs.com](https://www.silabs.com).

This application note discusses differences among the Series 1 and Series 2 EFR32 Wireless MCU generations that are relevant to proprietary wireless applications. "Proprietary" refers to applications based on RAIL (Radio Abstraction Interface Layer) or Connect, and typically implementing a **custom** radio configuration not defined by any wireless protocol standard. The target audience for this perspective consists of developers who are migrating existing EFR32 proprietary wireless applications to alternative EFR32 devices. This insight is also of interest to developers of new EFR32 proprietary solutions. However, this document is not intended to guide the migration of existing proprietary applications from non-EFR32 Silicon Labs wireless devices (EZRadioPRO, Si10xx, EZR32xG, and so on).

For developers migrating from EFR32 Series 1 to Series 2, [AN918.2: Series 1 to Wireless Gecko Series 2 Compatibility and Migration Guide](#) offers essential and extensive guidance for this endeavor from a platform perspective. Proprietary wireless applications (on any EFR32 device) additionally rely on RAIL or Connect as well as the Radio Configurator (a tool within Simplicity Studio) to manage and initially configure the integrated radio transceiver.

At a high level, RAIL and the Radio Configurator abstract away most differences that exist between EFR32 devices. The Connect stack and application framework leverage RAIL implicitly, and as such also benefit from (and further extend) abstraction to the application from most variation across potential target EFR32 devices. However, some device capabilities vary enough - or some features are unique enough - that they cannot simply be normalized "under the hood" of RAIL and/or the Radio Configurator. In these cases, customers must consider the impact of these generation-specific differences when planning a migration path for EFR32 proprietary wireless applications. Often, these distinguishing characteristics may in fact be enhancements that invite the porting of existing proprietary applications to new, more capable, EFR32 devices. More information on these differences relevant to proprietary wireless applications is presented in the following sections:

- [Minor Differences between EFR32 Generations](#)
- [RF Sense on EFR32xG22](#)
- [EM1P](#)

Note: This application note focuses on device-specific variations and where to find information on how RAIL accommodates them. As stated above, Connect abstracts most of these differences away. In the few cases

where it is appropriate, they will be highlighted in this document.

Minor Differences Between EFR32 Generations

Minor Differences between EFR32 Generations

This section presents a brief collection of generation-specific EFR32 differences that can impact proprietary wireless applications. Specific references for further reading are provided in each section. For more detailed information, see the [RAIL API Reference](#) as well as the reference manual and datasheet for your device.

RAIL (Radio) Configuration Compatibility

RAIL configures the radio on EFR32 devices with a desired PHY using one of two techniques:

- Load a static radio configuration generated for your project by the Simplicity Studio wireless application workflow.
- Load a RAIL embedded PHY using one of the protocol-specific APIs.

Static RAIL configurations are compatible across EFR32 families. All device families in a single generation share a common radio, but the static configurations are not compatible across EFR32 generations. In other words, for example, EFR32MG12 and EFR32FG12 share the same radio and therefore the same RAIL radio configuration, but a radio configuration generated for EFR32xG12 is not compatible with an EFR32xG14 device. When using the embedded PHYs, RAIL abstracts away most underlying differences, but in some cases, generation-specific limitations remain, as described in the following sections.

Connect may use the same static configuration files, hence it has the same compatibility limitations. The embedded RAIL configurations are available from Gecko SDK Suite version 3.1 onwards when using Connect.

Note: Static radio configurations are not currently supported on the EFR32xG21 platform. Hence, migrating to EFR32xG21 is only currently viable for applications using the embedded PHYs.

For more information on static radio configurations, see [Radio Configuration](#), and on embedded configuration routines, see [Protocol Specific](#) in RAIL.

Power Amplifier (PA)

Some PAs might not be available on certain parts. For example, devices with limited TX power might not have the high-power PA, or "sub-GHz only" parts won't have 2.4 GHz PAs. Beyond these OPN-specific limitations, PA options differ by EFR32 generation as follows:

- Sub-GHz PAs are only available on EFR32xG1, xG12, xG13, xG14, and xG23 generations.
- The EFR32xG1, xG12, xG13, xG14, and xG22 generations have both low-power and high-power 2.4 GHz PAs (the number of power levels per PA varies in xG22 vs xG1x).
- EFR32xG21 devices have low-power, mid-power, and high-power 2.4GHz PAs.
- EFR32xG23 devices have either a mid-power (14 dBm) or a high-power (20 dBm) sub-GHz PA, based on the device's OPN.
- EFR32xG24 devices have either a mid-power (10 dBm) or a high-power (19.5 dBm) 2.4 GHz PAs, based on the device's OPN.
- EFR32xG25 devices have a high-linearity PA dedicated for OFDM modulation, and a high-efficiency PA, both operating only on sub-GHz with a maximum power of 16 dBm.

For more info on EFR32 Power Amplifier (PA) Initialization, see [EFR32](#), on Power Amplifier treatment in RAIL, see [Power Amplifier](#), and on the available PAs, see the device's datasheet.

Antenna Switch

- EFR32xG21 has an integrated antenna switch supporting two RF paths.

- All other EFR32 devices have a single RF path, though external RF switches can be used.

For more information on antenna control in RAIL, see [Antenna Control](#).

RAIL Time Base

- The EFR32xG1 time base tick is 2 μ s.
- For all other EFR32 generations, the time base tick is 0.5 μ s.

For more information about the EFR32 RAIL timebase, see [EFR32](#), and about system timing in RAIL, see [System Timing](#).

Low-Frequency Timer Dependencies

RAIL does not use any low-frequency clocks by default, but LF clocks and a PRS channel are required when timer sync is enabled through `RAIL_ConfigSleep()`.

- EFR32xG1, EFR32xG12: An RTCC compare channel is required for RAIL when timer sync is used. This is channel 0 by default, but it can be customized in the application.
- All other EFR32 generations: The radio has a dedicated RTC timer for timer sync.

For more information on RAIL usage of EFR32 low-frequency clocks, see [EFR32](#), and on the PRS and RTCC channel customization, see [Sleep](#).

Buffer Handling

- Beginning with EFR32xG22, the RX and TX buffers must be word-aligned.

For more information on EFR32 Receive and Transmit FIFO Buffers in RAIL, see [EFR32](#).

IEEE 802.15.4 Options

- EFR32xG1 has limited 15.4g/e support compared to newer parts:
 - 4B CRC is not supported.
 - Only whitened packets are supported in 15.4g mode.
 - Enhanced ACKs are not supported.
 - Cannot receive MultiPurpose frame types.

For more information on IEEE 802.15.4 support in RAIL, see [IEEE 802.15.4](#).

Calibrations

- EFR32xG1 performs temperature (VCO) calibration when crossing 0oC (with 5oC hysteresis).

For more information on EFR32 radio calibration support, see [EFR32](#), and on radio calibrations in RAIL, see [Calibration](#).

Listen Before Talk (LBT)

- EFR32xG1 always averages RSSI during Clear Channel Assessment (CCA).
- All other parts can select between RSSI averaging and peak measurement.

For more information on the relevant RAIL option, see [Transmit](#) in the online RAIL documentation.

DMP Transition Time

In a Dynamic Multiprotocol (DMP) application, the time required for the radio to switch protocols varies as follows:

- Series 1 devices use 430 μ s transition time by default.
- Series 2 devices use 510 μ s transition time by default except EFR32xG21, which uses 500 μ s.

For more information on understanding the protocol switch time, see [RAIL Multiprotocol](#).

Direct Mode

TX and RX Direct Modes are available for 2-level modulations on most EFR32 Wireless MCU devices, except on EFR32xG21 and EFR32xG22.

- EFR32xG23 introduces direct to buffer mode.
- EFR32 Series 1 devices have better sensitivity results in packet mode compared to the direct mode when OOK modulation is used.

Wi-SUN Capabilities

EFR32xG25 devices are the first EFR32 Wireless MCU devices that support OFDM PHYs defined by the Wi-SUN FAN 1.1 standard, and SUN-OQPSK (also known as MR-OQPSK) defined by IEEE 802.15.4. The transceiver has a software-defined modem running on RISC-V architecture supporting OFDM modulation.

The Wi-SUN FAN 1.0 is supported by EFR32xG25, EFR32xG12, and EFR32xG23 device generations. However, only EFR32FG25 and EFR32MG12 support the Wi-SUN stack provided by Silicon Laboratories.

The radio has a dedicated Power Amplifier module optimized for Wi-SUN OFDM.

Note: Concurrent listening on EFR32xG25 devices for FSK and OFDM signals is supported, but once a signal is detected, only one can be demodulated.

RFFPLL on EFR32xG25

EFR32xG25 device generation introduces a new clock source module called Radio Frequency Friendly Phase Locked Loop (RFFPLL). RFFPLL generates a roughly 100 MHz clock from the XO frequency which is then used for driving the radio to avoid clock spurs in the RF chain. The actual configuration depends on the radio frequency band and the XO frequency.

RFFPLL settings are controlled via the Device Init: RFFPLL component, and can be set via the `CMU_RFFPLLInit()` API. RFFPLL must be configured during the initialization and cannot be changed at runtime. When creating a radio PHY, the RFFPLL band selection (**RF Frequency Planning** field) and the HFXO frequency must match the configuration applied via the platform code. If a PHY is assuming a certain RFFPLL band but the RFFPLL is configured differently, a RAIL assert error will be generated.

Note: RFFPLL can be optionally used as the system clock.

RF Sense On EFR32xG22

RF Sense on EFR32xG22

RF Sense is a low-power feature available on EFR32 Wireless MCU devices, except on EFR32xG23. With it, the radio can sense the presence of RF energy and "wake up" an MCU from EM2 (or any other) power mode. In practice, RF Sense provides an ultra-low power interrupt source that runs on the ULFRCO clock.

Since the RFSense block implements a wide band circuit, it can detect energy in a broad frequency range between 100MHz and 5GHz - filtered only by the matching network of the RF front end. On one hand this is an advantage, as there is no need for additional PCB components to support the feature. However, there is a drawback: the wake-on-RF capability is responsive to any unfiltered interferer signals.

EFR32xG22 devices include an enhanced RF Sense module, which improves performance as compared to the EFR32 Series 1 implementation in multiple ways:

- RF Sense works below 0oC.
- RF Sense works even when voltage is scaled down.
- In addition to legacy behavior, the EFR32xG22 introduces "Selective Mode" RF Sense.

Legacy Mode

In Legacy (Energy Detection) mode, EFR32xG22 RF Sense is fully compatible with the feature in Series 1 devices. This means that if the RF Sense module detects energy for a configured duration, it generates an interrupt.

Selective Mode

Selective mode mitigates the unfiltered nature of RF Sense. Instead of simply detecting energy for a given time period, it detects "a pattern of energy" - which is essentially an On-Off Keying (OOK) packet. The packet is Manchester-coded and uses a fixed 1 kbps bitrate, 1 B preamble, and 1-4 B sync word (no payload is added, see additional details below). This packet can be transmitted by any 2.4 GHz OOK-capable device, including most EFR32 wireless MCUs.

Setting Up Selective RF Sense

Selective RF Sense can be started with the API `RAIL_StartSelectiveOokRfSense()` instead of the legacy `RAIL_StartRfSense()`. The API takes a `config` parameter of type `RAIL_RfSenseSelectiveOokConfig_t` that sets up the desired RFSense Selective mode sync word configuration. After this function is called, the interrupt is enabled and the device can enter sleep mode.

On the transmit side, calling `RAIL_ConfigRfSenseSelectiveOokWakeupPhy()` switches the radio config to the special wakeup PHY. Next, configure the wakeup packet using `RAIL_SetRfSenseSelectiveOokWakeupPayload()` so that it matches the configuration on the RX side. Finally, use `RAIL_StartTx()` (or any other TX API) to send the wakeup packet. See [Selective Mode Transmit Example Without Using API](#) for an illustration.

Selective RF Sense (RX and TX) can be performed in RAILtest as well, see [UG409: RAILtest User's Guide](#) for details.

Note: On EFR32xG21 devices, you cannot transmit the Wakeup Packet due to the lack of OOK support.

The Wakeup Packet

The wakeup packet is a fixed-configuration OOK packet with the following settings:

- Starts with a 1 B preamble (always 0x55).

- Followed by a 1-4 B sync word.
- No payload required.
- Both the preamble and sync word are transmitted LSB first.
- 1 kcps chiprate (Manchester coded)/500 bps bitrate (useful data rate).
- Recommended carrier is 2.45 GHz (EFR32 radios use this frequency).

Selective Mode Transmit Example Without Using API

Assume you select `0xb16e` as your sync word, and you want to transmit it with only a signal generator (or a simple radio with MSB-first byte handling and no Manchester coder).

First, flip the endianness of both preamble and sync word: `0x55` becomes `0xaa` and `0xb16e` becomes `0x768d`.

The full packet is then `0xaa768d`, which after Manchester coding becomes `0x99996a6995a6`.

Configuring this encoded packet and transmitting on 2.45 GHz with high enough TX power should wake up a device configured for selective RF Sense with the `0xb16e` sync word.

EM1P

EM1P

Beyond the energy modes available on prior devices, EFR32xG22 introduces **EM1P**, a new method to reduce energy consumption while using the radio. In this (intermediate) mode, the HF crystal oscillator (HFXO) is kept running (and the radio remains active) if the following conditions are met:

- The radio state is anything other than idle (for example RX or TX), and
- Power Manager or the application software requests to enter EM2.

This sequence puts the Cortex-M33 into sleep mode, and clocks to the core, and all high-speed peripherals are disabled (peripherals and oscillators capable of EM2, EM3, or EM4 operation remain available).

On earlier EFR32 devices, entering EM2 mode would unconditionally shut down the radio, regardless of the operation in progress. Doing so during certain radio activities would potentially incur undesirable side effects (for example, FIFO corruption).

Comparison of EM1P and Other Energy Modes

EM1P is not technically a standalone energy mode (like EM1, EM2, and so on). Rather, it is an operating condition where most of the EFR32 enters EM2 Deep Sleep, but the radio (and its requisite HFXO clock source) are retained. As such, EM1P can be (perhaps confusingly) viewed as "EM2 with Radio".

This combination results in lower current consumption than EM1 mode, and higher consumption than full EM2. The following table depicts some observations of these influences on energy usage taken on a single device. Current consumption was measured initially at EM0, when the radio was idle and in RX. After transitioning into EM1/EM2, current was again measured. The cell in the bottom-right corner represents the EM1P measurement result (radio was in RX, device then transitioned to EM2).

Radio State	Initial (EM0) Current	Final Energy Mode	Resulting Current
Idle	2.36 mA	EM1	1.60 mA
"	"	EM2	1.56 µA
RX	5.92 mA	EM1	5.15 mA
"	"	EM2	4.85 mA

Note: The values above are **not** guaranteed, and will vary across devices and scenarios due to numerous factors. They are presented here as a single illustrative example of the general performance trends among these different modes of operation. Consult your device datasheet for all definitive guidance regarding specifications and performance.

Scheduled Radio Operations

EM1P mode does not impact scheduled RX and scheduled TX operations, as in those cases the radio is in the idle state when the firmware enters EM2.

RAIL Fundamentals

RAIL Fundamentals

NOTE: This section replaces *UG103.13: RAIL Fundamentals*. Further updates to this user guide will be provided here.

The Silicon Labs RAIL (Radio Abstraction Interface Layer) library provides an intuitive, easily-customizable radio interface layer that supports proprietary or standards-based wireless protocols. RAIL is designed to simplify and shorten the development process. Developers no longer have to deal with hundreds of registers across multiple products, but can instead rely on a unified software API. RAIL, delivered through the Silicon Labs Flex SDK (Software Development Kit), also makes applications portable across Silicon Labs wireless products.

Silicon Labs' *Fundamentals* series covers topics that project managers, application designers, and developers should understand before beginning to work on an embedded networking solution using Silicon Labs chips, networking stacks such as EmberZNet PRO or Silicon Labs Bluetooth, and associated development tools. These guides can be used as a starting place for anyone needing an introduction to developing wireless networking applications, or who is new to the Silicon Labs development environment. Proprietary is supported on all EFR32FG devices. For others, check the device's data sheet under Ordering Information > Protocol Stack to see if Proprietary is supported. In Proprietary SDK version 2.7.n, Connect is not supported on EFR32xG22.

Even with the many wireless standards available, proprietary wireless is still the only option to communicate with legacy, proprietary networks. Also, using a wireless standard is always a compromise—a protocol designed for the application can be much better optimized for energy consumption, data throughput, or range.

However, the flexibility of proprietary protocols comes with a price. It is usually more difficult to develop such protocols, and they are usually incompatible with newer infrastructures. Security design is especially difficult and this must be considered when designing a proprietary protocol.

To develop a proprietary protocol, you need direct control over the radio hardware. However, a general purpose, multiprotocol capable radio, like the Wireless Gecko (EFR32™) is very complex, and would take months to understand. To shorten this process, Silicon Labs provides:

- Radio Configurator, a tool that can generate a radio configuration from a few input parameters like frequency and bitrate.
- RAIL (Radio Abstraction Interface Layer), a C library, which provides a much simpler interface to control the radio from the application code.

Silicon Labs RAIL is the lowest layer for all networking stacks developed internally by Silicon Labs, as well as by the company's customers and third-party partners. RAIL supports a diverse set of radio configurations and functionality and is one of the key underlying technologies of Silicon Labs wireless products.

To make software portability as simple as possible, RAIL was developed with the following goals:

- RAIL API should be backward compatible within major versions. A code developed for RAIL 2.0 should run with the newest version of the RAIL 2.x release cycle.
- RAIL API should be compatible between supported parts as much as possible. Although there are a few APIs that are not available on all parts, general usage—like receive, transmit and state transitions—should be the same.

RAIL Overview

RAIL Overview

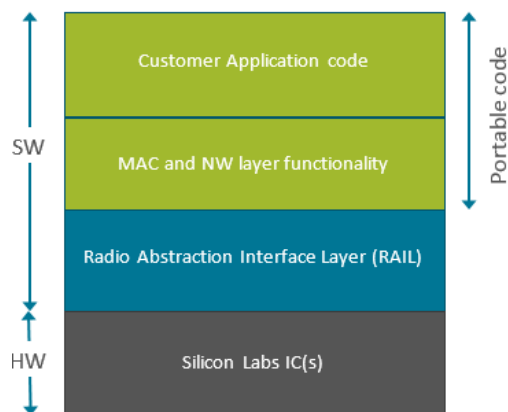
The RAIL library itself is delivered through the Gecko SDK Suite, but the supporting tools and example applications are also part of the Flex SDK. In the Flex SDK developers can develop protocols that work directly with RAIL, or configure applications based on the Silicon Labs Connect stack. The Silicon Lab Connect stack provides a fully-featured, easily-customizable wireless networking solution optimized for devices that require low power consumption and are used in a simple network topology. For more information, see [Silicon Labs Connect Fundamentals](#).

RAIL components in the Flex SDK are:

- The RAIL library: Provides a programming interface to radio functionality, as shown in the following figure.
- The Radio Configurator: Part of Simplicity Studio, a calculator and interface that allows developers to configure static parameters of the radio physical layer. For details, see *AN971: EFR32 Radio Configurator Guide for RAIL in Simplicity Studio v4* or *AN1252: EFR32 Radio Configurator Guide for Simplicity Studio v5*.
- RAILtest, a sample application that includes a serial command for each RAIL library feature, to allow scripted testing and ad hoc experimentation. RAILtest can be built with any PHY, including 802.15.4 and Bluetooth Smart. Many of the RAILtest serial commands can be used for lab evaluation. RAILtest is also a good starting point for testing various RAIL features and can be used as a reference implementation of many APIs.
- Other example applications: Can be used as is for evaluation and also serve as a starting point for application development.
- Documentation, delivered through Simplicity Studio.
- Components built on top of RAIL, that provide basic functionality, like initialization, transmit or receive.

For more information about using the example applications in the Flex SDK to begin development with RAIL, see *QSG138: Proprietary Flex SDK v2.x Quick Start Guide* or *QSG168: Proprietary Flex SDK v3.x Quick Start Guide*.

Although the RAIL library supports the complete EFR32 portfolio, the radio configurator and the Flex SDK examples are only available on EFR32FG devices and other EFR devices with Proprietary support. Check the device's data sheet under Ordering Information > Protocol Stack to see if Proprietary is supported.



RAIL Library

RAIL Library

RAIL functionality is delivered as a library that is linked to the developer's application. The RAIL library implements the core features and runtime APIs needed to configure and control the radio.

The RAIL library works by taking intuitive and easy-to-use commands from the RAIL API and translating them into register-level code used to control radio and communications functions. The API commands remain constant across ICs. The changes in the underlying code are transparent to the developer or system tester. This also allows developers to create multiple stacks for different products quickly, as they are always presented with a similar software radio interface. RAIL provides the foundation upon which developers can implement their own MAC layer and network layer functionality.

Where possible, all features currently implemented for the EFR32 will be implemented for future ICs, allowing for easy migration of all RAIL-based applications. The main differences between EFR32 generations are highlighted in [EFR32 Migration Guide for Proprietary Applications](#).

The RAIL API is built up from commands and events. Commands can be used to initiate something from the software, while events are generated by the hardware (usually from an interrupt) to let the software know of something, like a received frame.

RAIL is built to support multiple protocol stacks at the same time. The Silicon Labs Dynamic Multiprotocol solution is implemented with the Radio Scheduler, a preemptive, priority scheduler, which can schedule the radio hardware between protocols. The API is the same for single-protocol and multiprotocol RAIL, but multiprotocol RAIL applications should be implemented somewhat differently. See the [Dynamic Multiprotocol User's Guide](#) for more information.

The library supports GCC and IAR compilers for ARM.

RAIL Features

RAIL Features

RAIL includes the following features in software, as well as many others. This table reflects the most recent version of RAIL. If you are interested in when a particular feature became available, see the RAIL changelog.

RAIL Highlights	Description
Simple Transmit and Receive	Both transmit and receive can be started with a single API. Finished transmission and received packets are signaled with RAIL events.
Automatic State Transitions	Configurable transitions to go from one state to the next based on successful or unsuccessful transmit and receive operations. These support an optional delay as well to allow for consistent timings from one state to the next.
Frame Buffering	Two configurable size frame buffers—one for receive and one for transmit—are available. Both can hold multiple frames.
Timekeeping, Timestamping, and Timers	RAIL automatically maintains a RAIL timebase with μ sec granularity. It can be used for:
"	Timestamping packets (enabled by default) at different packet positions
"	Scheduling receive and transmit operations
"	One general purpose timer with optional timer virtualization in RAIL (called MultiTimer)
"	Optional synchronization with RTC, to keep time and make scheduling possible from EM2 sleep mode
Scheduled Transmit	Transmit at an absolute or relative time in the future.
Scheduled Receive	Start receiving at an absolute or relative time in the future with an optional timeout after that start time.
CCA (Clear Channel Assessment) with Retransmission	Supports two common medium access methodologies, which delays transmission until the channel is clear:
"	CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) -- based on IEEE 802.15.4 specification.
"	LBT (Listen Before Talk) -- based on ETSI EN 300 220-1 specification.
RSSI (Received Signal Strength Indicator) Read	Ability to read instantaneous and average RSSI values from hardware. RSSI and Link Quality Indicator (LQI) measurements for each received frame is also available.
Address Filtering	Two address fields are supported (with fixed location), both maximum four bytes long. Each field can filter for four addresses (for example, unicast and broadcast address). Addresses can be enabled and configured at runtime. For IEEE 802.15.4, this is handled specially, since it uses addresses with the location configured with the frame type.

RAIL Highlights	Description
Auto ACK	Automatic transmission of a response packet after some delay to every successful receive. This can be a constant payload or dynamically loaded by the user any time before the transmission is supposed to begin. For IEEE 802.15.4 this is handled automatically to improve performance.
IEEE 802.15.4 Helper Features	These features include:
"	Address filtering (supporting three short address, long address and PAN ID)
"	Auto ACK
"	Event on Data Request command to set the frame pending bit
"	Note: To implement IEEE 802.15.4-based protocols, Silicon Labs recommends using the Connect stack's MAC mode, which is a full IEEE 802.15.4 MAC layer implementation. For more information, see <i>UG235.03: Architecture of the Silicon Labs Connect Stack v2.x</i> or <i>UG435.03: Architecture of the Silicon Labs Connect Stack v3.x</i> .
Bluetooth LE and Z-Wave Helper APIs	Helpers for implementing a Bluetooth LE or Z-Wave networking stack. These provide special modes and optimizations in hardware for these specific modes.
Multi-PHY Support	The PHY configuration can be changed runtime. Also, RAIL can load PHY.
Multiprotocol Support	Separate version of the RAIL library with a radio scheduler that can manage requests from different radio configurations on the same chip:
"	Supports time-slicing N different radio configurations where $N \geq 2$.
"	Allows for different priority tasks from the different configurations and manages them so the highest priority is always completed.
"	Allows for tasks to be moved around if the timing isn't critical, for most robust network.
Calibration Support	Chip-specific calibrations. Commonly these include recalibrating for large temperature changes while in receive and image rejection calibration for increased sensitivity.
Stream Generation for RF Testing	RAIL can generate Continuous Wave and PN9 pseudo-random modulated streams with any PHY for RF testing.
Receive Antenna Diversity	Allows toggling between two antennas in receive mode and selecting the one with the stronger signal to improve sensitivity. Note that this is only supported on certain chips and PHY configurations.
RAIL API to Get Entropy from Radio	Radio can be used to collect entropy from receive chain noise.
Power Manager Support	When enabled, notifies the power manager component when the radio is on to prevent Energy Mode 2 (EM2) sleep.

Major Updates to RAIL

Major Updates to RAIL

RAIL 2.x added many new features, including Dynamic Multiprotocol, and provided a cleaner, more understandable API compared to RAIL 1.x. For more information, see [AN1113: Porting RAIL Applications to RAIL Version 2.x](#).

Gecko SDK 3.x still includes RAIL 2.x so the API remains backwards-compatible. The new framework simplifies project migration and provides many more reusable components for general tasks such as initialization. However, the RAIL API itself did not change; it only added a few new features. For more information, see *AN1254: Transitioning from the v2.x to the v3.x Proprietary Flex SDK*.

Next Steps

Next Steps

See *QSG138: Proprietary Flex SDK v2.x Quick Start Guide* or *QSG168: Proprietary Flex SDK v3.x Quick Start Guide* for instructions on how to install Simplicity Studio and get started using RAIL from the Flex SDK.

To get more familiar with software development in RAIL, see the [API documentation](#) and the Silicon Labs [RAIL tutorials](#).

Overview

RAIL Developers Guide

These pages provide details for developers who wish to include RAIL functionality in their applications. The focus of these pages is the firmware development, including basic usage, optimizing for timing and/or power consumption, and debugging techniques.

Content includes:

- [Using RAIL for IEEE 802.15.4 Applications with EFR32](#): Describes using the Flex SDK for 802.15.4 development on EFR32 Wireless parts.
- [Power Amplifier Power Conversion Functions in RAIL 2.x and Higher](#): Outlines how to account for the variation in output characteristics across custom boards and applications for the Silicon Labs EFR32 family of chips.
- [Detailed Timing Test Results for RAIL \(AN1392\)](#): Provides information about various timing measurements that may be of interest when using RAIL to develop an application.
- [Direct Mode](#): Outlines the Rx Direct Mode feature of the EFR32 Wireless Gecko family.

Using RAIL for IEEE 802.15.4 Applications with EFR32

Using RAIL for IEEE 802.15.4 Applications with EFR32

This section replaces *AN1365: Using RAIL for IEEE 802.15.4 Applications with EFR32*. Further updates to this application note will be provided here.

This application note describes using the Flex SDK for 802.15.4 development on EFR32 Wireless parts. It includes features and limitations, using the radio configurator and supplied software components, and a description of the examples.

Proprietary is supported on all EFR32FG devices. For others, check the device's datasheet under Ordering Information > Protocol Stack to see if Proprietary is supported. In Proprietary SDK version 2.7.x, Connect is not supported on EFR32xG22.

Overview

RAIL provides numerous HW-accelerated IEEE 802.15.4 compliant MAC features, such as format filtering, address filtering, auto ACK-ing, and filtering based on the frame type. The MAC features are available if the radio is configured via official IEEE 802.15.4 PHYs loaded by certain RAIL APIs.

However, as RAIL exhibits only a limited subset of the standard PHYs defined by the 802.15.4 PHY layer specification, it is possible to define additional PHYs in the Radio Configurator interface compatible with the IEEE 802.15.4 standard on which the MAC features can also be enabled.

Configure 802.15.4 PHYs In Rail

Configure 802.15.4 PHYs in RAIL

This section describes how to configure EFR32 radios with an IEEE 802.15.4 compatible PHY. Other PHYs might not be available in RAIL, but can be selected in the Radio Configurator.

Note: For the available PHYs and 802.15.4 specific RAIL features on the selected radio platform generation, please consult:

- the device's datasheet
- the Radio Configuration GUI
- the RAIL's documentation

2.4 GHz 250 kbps O-QPSK PHYs

The 2.4 GHz 250 kbps O-QPSK PHY is the most commonly used instance of the IEEE 802.15.4 PHYs supported in RAIL as an embedded configuration. Zigbee and Thread applications (indoor PAN network protocols) also configure this PHY when initializing the stack.

To configure this PHY, call the `RAIL_IEEE802154_Config2p4GHzRadio()` RAIL API. This procedure shall take the place of `RAIL_ConfigChannels()`.

This PHY initializes the radio for 2.4 GHz operation as defined by IEEE 802.15.4-2011 section 8.1.2.2.

This PHY uses a 1-byte header (PHR) where the 7 LSB bits determine the number of bytes in the PSDU (payload data).

RAIL supports this PHY along with each combination of the following options:

- Antenna Diversity
- Radio Coexistence
- FEM

Configure these PHYs via the `RAIL_IEEE802154_Config2p4GHzRadio[AntDiv][Coex][Fem]()` API variants.

These PHYs are highly optimized and validated configurations not available in the Radio Configurator.

Note: In these PHY configurations, only channels 11-26 are available. This PHY should not be confused with the SUN O-QPSK PHY. The SubGHz O-QPSK PHYs are documented in a subsequent section.

Connect 2.4 GHz O-QPSK PHYs

The Radio Configurator provides the `Connect 2.4GHz OQPSK 2Mcps 250kbps` PHY, which is mostly compatible with 802.15.4, but might not meet all requirements, and might fail regulatory requirements (such as FCC). This PHY has worse RF performance on most EFR32 radio generations than the PHYs introduced in the previous section.

However, configuring via the Radio Configurator can be useful if the user has some experience with the 802.15.4 packet format, but doesn't need to comply with the standard, as these can be customized in many ways (DSSS configuration, custom datarate, different syncword, and preamble configuration, channel number, and frequency assignment, etc.).

You can find more information in [UG435.02: Using Silicon Labs Connect v3.x with IEEE 802.15.4](#).

Wi-SUN and SUN PHYs

The Wi-SUN and SUN PHYs are available in the Radio Configurator. However, some of the PHYs below might have an alternative configuration available via RAIL APIs. Use the PHYs from the Radio Configurator to be fully compatible with the

standard.

SUN FSK

These PHYs can be found under the **Base** profile `-> IEEE 802.15.4 SUN FSK \*`.

The SUN FSK PHYs supported in the Radio Configurator always use `phySunFskSfd = 0` SFD configuration.

SubGHz GB PHYs

The following PHYs are being used most commonly in UK metering standards, for example, Zigbee. These can be configured via RAIL API calls:

- `RAIL_IEEE802154_ConfigGB863MHzRadio()`
- `RAIL_IEEE802154_ConfigGB915MHzRadio()`

Note: These PHYs provide quasi SUN-FSK configuration but are not fully compliant with that standard (there are limitations on the PHY header's construction, e.g., the maximum payload length is limited to 127 bytes in these PHYs).

The available channels for the 863 MHz PHY ranges between 0xE0..0xFA, while for the 915 MHz PHY, these are 0x80..0x9A, 0xA0..0xA8, 0xC0..0xDA.

SUN O-QPSK

These PHYs can be found under the **SUN OQPSK** Profile.

Wi-SUN specific PHYs

- The HAN PHYs can be found under the **Wi-SUN HAN** Profile.
- The FAN PHYs can be found under the **Wi-SUN FAN 1.0** Profile or **Wi-SUN FAN 1.1** Profile.

PHYs are using the 802.15.4 frame format

Using the PHYs described in the following section, the radio might not be fully 802.15.4 compatible and could fail on regulatory and/or protocol certification. The 802.15.4 frame format is a well-known and common specification. Using such a radio configuration might accelerate the product development cycle, and it makes available the 802.15.4 specific RAIL APIs.

If the PHY you want to use is supported by RAIL, Silicon Labs recommends using it from there, as these are validated and fine-tuned through config options that might not be available on the radio configurator.

If you select any PHY from the **Connect** Profile and **Long Range** Profile in the Radio Configurator, most of the payload-related customization becomes unavailable because these PHYs ensure 802.15.4 frame format compatibility, but overall, these PHYs are not fully compatible with the standard.

However, in cases where you need more customization options available in the Radio Configurator, you might have set up 802.15.4 frame format compatibility as it is discussed in the following section.

Build Custom PHYs in the Radio Configurator

In cases where you need full control over the PHY customization, you need to set it up directly in the Radio Configurator.

To set up the base of the config, follow these steps:

- Disable all **Advanced** options.
- Set up modulation and frequencies as per the requirements of the standard.
- Configure channel spacing and channel numbering according to the standard.
- Preamble and Sync word (SFD) depends on the PHY format; set it up accordingly.

- Use LSB first everywhere unless stated otherwise. (*)

(*) SUN-FSK is an exception, which defines the length field MSB first. The length can be handled by the hardware, but you must keep in mind that you have to reverse the endianness of these bytes for both RX and TX in the application. Use the following code to set and get the length of the packet:

```
// length helper functions for Wi-SUN 15.4 frame formats
void setLength(uint8_t *payload, uint16_t length){
    payload[1] = __RBIT(length & 0xFF) >> 24;
    payload[0] = __RBIT(length >> 8) >> 24;
}
uint16_t getLength(uint8_t *payload){
    uint16_t length = __RBIT(payload[1]) >> 24;
    length |= (__RBIT(payload[0]) >> 16) & 0x0700;
    return length;
}
```

Variable length setup for 1Byte PHR with 7bit length

On 2.4 GHz O-QPSK, this is covered by the embedded config. Otherwise:

- Check `Header Enable`
- Set `Header Size` to `1`
- Set `Frame Length Encoding` to `VARIABLE_LENGTH`
- Set `Frame Bit Endian` to `LSB_FIRST`
- Check `Length Includes CRC Bytes`
- Set `Minimum Length` to `0`
- Set `Maximum Length` to `127`
- Set `Variable Length Bit Size` to `7`
- Set `Variable Frame Length Adjust` to `0`
- Set `Variable Length Bit Endian` to `LSB_FIRST`
- Set `Variable Length Bit Location` to `0`

Variable length setup for 2Byte PHR and 11bit length (SUN FSK)

- Check `Header Enable`
- Set `Header Size` to `2`
- Set `Frame Length Encoding` to `VARIABLE_LENGTH`
- Set `Frame Bit Endian` to `LSB_FIRST`
- Check `Length Includes CRC Bytes`
- Set `Minimum Length` to `0`
- Set `Maximum Length` to `2048`
- Set `Variable Length Bit Size` to `11`
- Set `Variable Frame Length Adjust` to `0`
- Set `Variable Length Bit Endian` to `MSB_FIRST`
- Set `Variable Length Byte Endian` to `MSB_FIRST`
- Set `Variable Length Bit Location` to `0`

Note: The default RX FIFO size is 512B. You might want to increase that through `RAIL_SetRxFifo()` or use FIFO mode to download the packet in chunks.

CRC configuration

2B (CCIT-16)

- Set `CRC Input Bit Endian` to `LSB_FIRST`
- Uncheck `CRC Input Padding`

- Set CRC Polynomial to CCITT_16
- Set CRC Seed to 00
- Set CRC Output Bit Endian to MSB_FIRST
- Set CRC Byte Endian to MSB_FIRST
- Uncheck CRC Invert
- Uncheck CRC Header

4B (ANSI X3.66-1979)

- Set CRC Input Bit Endian to LSB_FIRST
- Check CRC Input Padding
- Set CRC Polynomial to ANSI366_1979
- Set CRC Seed to 0xFFFFFFFF
- Set CRC Output Bit Endian to MSB_FIRST
- Set CRC Byte Endian to MSB_FIRST
- Check CRC Invert
- Uncheck CRC Header

Whitening (2B PHR only)

- Set Whitening Output Bit to 0
- Set Whitening Seed to 0x01F0
- Set Whitening Polynomial to PN9
- Uncheck Whiten Header

Best Practices for Configuring the PHY

- If the selected PHY is available via RAIL API calls, use that (except SUN and Wi-SUN PHYs).
- If the selected PHY is available via the Radio Configurator, use that.
- If the selected PHY isn't created yet, you can configure it on your own on the Radio Configurator GUI. Note that in this case, the protocol configurator might not meet the specification.

802.15.4 HW Accelerated MAC Features

802.15.4 HW Accelerated MAC Features

The APIs introduced in the following chapters can work with any proprietary PHY (just like any PHY defined under the Connect Profile in the Radio Configurator) which meets the 802.15.4 frame format specification. This is the only requirement of the 802.15.4 PHY layer to use the protocol-specific RAIL APIs. The Connect SDK is a great example of how to use the protocol specific features on PHYs that are not fully 802.15.4 compatible.

The hardware does support address and frame filtering, Auto-ACK, and support to handle the frame pending bit in the ACK, implemented in RAIL. To enable it, use the `RAIL_IEEE802154_Init()` API call and the other calls related to address filtering or frame pending.

You can configure the Address Filtering and Auto-Ack features via the Init API, though those features have individual configuration APIs introduced in the following chapters.

Address Filtering

To configure address filtering, call `RAIL_IEEE802154_SetAddresses()` with a structure containing all addresses, or call the individual `RAIL_IEEE802154_SetPanId()`, `RAIL_IEEE802154_SetShortAddress()`, and `RAIL_IEEE802154_SetLongAddress()` APIs.

You can set up the addresses in the `RAIL_IEEE802154_Init()` API, too.

Configuring the addresses packet reception will be immediately aborted if the configured Address Filtering does not match.

RAIL supports intra-PAN (network) communication using multiple addresses.

Note that IEEE 802.15.4 Address Filter is not the same as the generic RAIL's Address Filter feature.

Best Practices for Address Filtering

Broadcast addresses are supported by default without any additional configuration and do not consume extra address slots.

Set unused addresses to the broadcast address (`0xFFFF`) to avoid false detections.

Do not change addresses during reception. Reconfiguring the Address Filter while receiving a packet may result in undesired behavior.

For more details, see the Detailed Description at IEEE 802.15.4 documentation and `RAIL_IEEE802154_AddrConfig_t` type's documentation.

Auto-Ack and Frame Pending

Auto-ACK is controlled by the `ackConfig` and `timings` fields passed to `RAIL_IEEE802154_Init()`. After initialization, they may be controlled using the normal Auto-ACK and State Transitions APIs.

The `ackTimeout` field only covers sync word detection of the received packet. Therefore, the ACK's PHY header and payload time shall be accommodated to get the correct `ackTimeout` setting. Find an example at RAIL's IEEE 802.15.4 – Detailed Description.

To listen for an ACK packet, you must transmit the packet with the `RAIL_TX_OPTION_WAIT_FOR_ACK` option.

The ACK frame includes the same sequence number as the frame that it responds to.

The AR bit in the Frame Control Field specifies whether an acknowledgment is required from the receiver device. Upon a reception, the receiver sends an ACK frame only if the frame passes the filtering. If this bit is 0, the receiver does not send an ACK packet. Usually, the AR is located at the 5th position of the FCF.

The Frame Pending bit is used for data polling. The end device polls the coordinator with a data request command. If the coordinator has a message to the end device, it responds with an ACK packet with the frame pending bit set in the FCF.

When in IEEE 802.15.4 mode, the ACK will generally have a 5-byte length (FCF, sequence number and FCS from the MAC fields, but have no payload), its Frame Type will be ACK, its Frame Version 0 (2003), and its Frame Pending bit will be false unless the `RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND` event is triggered, in which case, it will default to the `RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks` setting. If the default Frame Pending setting is incorrect, the app must call `RAIL_IEEE802154_ToggleFramePending()` (formerly `RAIL_IEEE802154_SetFramePending()`) while handling the `RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND` event.

This event must be turned on by the user and will fire whenever a data request is being received or fully received so that the stack can determine if there is pending data. Note that if the default Frame Pending bit needs to be changed, it must be done quickly. Otherwise, the ACK may already have been transmitted with the default setting. To trigger this event earlier, during the packet reception, use the `RAIL_IEEE802154_EnableEarlyFramePending()` API.

Check the return code of `RAIL_IEEE802154_ToggleFramePending()` to be sure that the Frame Pending bit was changed in time.

The `RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND` is triggered only for Data Request MAC command frames requesting ACK. To trigger this event also for Data type frames call the `RAIL_IEEE802154_EnableDataFramePending()` API.

Enhanced ACK

The `RAIL_IEEE802154_E_OPTION_ENH_ACK` option allows the radio to receive packets with Frame Version 2 in the MAC header, as RAIL normally accepts packets with Frame Version 0 or 1. Enabling this option is required for handling Enhanced ACK frames.

RAIL assembles Immediate ACK frames automatically, but not Enhanced ACK frames. A dedicated RAIL event, `RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND`, is implemented to parse a partly-received frame to determine the type of ACK needed to provide enough time to generate the ENH-ACK frame.

Note that this feature does not enable receiving Multipurpose frames. It can be enabled via the `RAIL_IEEE802154_AcceptFrames()` API.

Note: Thread protocol always sets the FP bit to 1.

Implicit Broadcast

The `RAIL_IEEE802154_E_OPTION_IMPLICIT_BROADCAST` option enables the MAC implicit broadcast feature. If the Frame Version is 2 and the frame lacks the destination PAN ID and address, then the packet is treated like a broadcast packet. If this option is disabled, such a packet is usually filtered.

Runtime PHY Reconfiguration with SUN FSK and Wi-SUN PHYs

2B Header Options

Call `RAIL_IEEE802154_ConfigGOptions()` with the `RAIL_IEEE802154_G_OPTION_GB868` option to support dynamic CRC and whitening reconfiguration coded in a 2B PHR PHY.

In this mode, RAIL automatically reads the PHR field of the transmitted and received packet, and reconfigures the radio (i.e., enables/disables whitening for the rest of the payload, and selects 2/4-bytes CRC configuration) according to the FCS and WHITENING bitfields during the ongoing operation.

Though the API's name is a bit misleading, this enables all the 802.15.4g available features.

Dynamic FEC

To enable/disable 802.15.4g dynamic FEC feature use the `RAIL_IEEE802154_G_OPTION_DYNFEC` option.

SFD field (implemented as the syncword) indicates whether the rest of the packet is FEC encoded or not. As the SFD field is configured as the syncword on the EFR32 radio platform, it enables dual syncword detection to detect whether FEC should be enabled or not.

Address Filtering and FEC work only simultaneously on radio platforms where `RAIL_IEEE802154_SUPPORTS_G_DYNFEC` is defined. That means on former chips, Promiscuous mode should be enabled to support FEC.

Dynamic FEC configuration is supported on SUN FSK PHYs only.

Find more information at [IEEE 802.15.4](#).

Mode Switch

The mode switch mechanism enables a device using the MR-FSK PHY to change its symbol rate and/or modulation scheme on a packet-by-packet basis. This feature can be enabled/disabled with the `RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH` option.

A specific mode switch PPDU is used to inform the receiver of the mode switch and specifies the new PHY mode of the following PPDU.

Switching to a new PHY mode with a higher rate typically by sending/receiving a specific Mode Switch packet that indicates the incoming new PHY mode. The Mode Switch packet is an FSK-modulated 2-byte PHY header with no payload.

Find more information at [IEEE 802.15.4](#).

Note that this is the Mode Switch feature defined by Wi-SUN and not the 802.15.4g standard.

Examples

Examples

The following examples demonstrate basic functionalities of the 802.15.4 RAIL features. Usually, these tests require 2 RAILtest nodes, one operating as a transmitter and the other as the receiver. Issue the commands on both nodes if not stated otherwise.

For the sake of simplicity, we demonstrate the 802.15.4 features using the in-built 2.4 GHz 250 kbps O-QPSK and the SubGHz GB PHYs.

Applying a PHY from RAIL

```
rx 0 // Configs can be loaded only in radio Idle state
config2p4GHz802154
// or
config863GHz802154
// or
config915MHz802154
rx 1 // Make the RX active
```

Transmitting a packet (tx 1) from TX node is received on the RX node.

The default payload of the RAILtest application is constructed so that the device will transmit a 16 bytes length frames. Therefore, loading the subGHz PHYs also changes the first two bytes of the payload (the PHR field) to comply with the default 16 byte length packet configuration of the RAILtest application, if the default payload has not been changed yet.

To set different payload lengths, see the next example.

Use the `getChannel` command to see which channel the radio is on.

Variable Length Payload

To change the payload length of the transmitted packet, the Frame Length (`FL`) field of the PHR must be changed. The Frame Length field specifies the total number of bytes in the PSDU (`PSDU_LEN`). RAIL automatically sets the length of the packet according to the Frame Length field on both TX and RX modes.

The length of the PHR (`PHR_LEN`) can be 1 or 2 bytes in these PHYs.

Keep in mind that CRC bytes must be included in the Frame Length field. 802.15.4 supports both 2 and 4 bytes length CRC sequences, so the length of the CRC (`CRC_LEN`) can be 2 or 4.

In the examples below, `PACKET_LEN` denotes the packet length RAIL uses (including the PHR but excluding the CRC), while `PSDU_LEN` is the value of the Frame Length field defined by 802.15.4. In other words, RAIL expects `PACKET_LEN` bytes in the TX FIFO before transmitting the packet.

$$\text{PACKET\_LEN} = \text{PSDU\_LEN} + \text{PHR\_LEN} - \text{CRC\_LEN}.$$

If `PACKET_LEN` is higher than the default RAILtest payload (16), the `setTxLength <MAX_PACKET_LEN>` must also be issued, where `MAX_PACKET_LEN` is the length of the maximum expected payload length to transmit. If `PACKET_LEN > MAX_PACKET_LEN` RAIL generates a `RAIL_EVENT_TX_UNDERFLOW` event resulting in packet abort as there are no more bytes stored in the TX FIFO to transmit.

For the 2.4 GHz PHY (1B PHR), the Frame Length field is the 7 LSB bits of the first byte.

```
setTxPayload 0x00 <FL[6:0]>
```

where FL can range between 4 - 127. The MSB bit shall be set to 0.

```
CRC_LEN = 2
PHR_LEN = 1

FL = PSDU_LEN = PACKET_LEN + 1
```

More examples:

```
rx 0

config2p4GHz802154

setTxPayload 0x00 0x04 // PACKET_LEN = 3; PSDU_LEN = 4; shortest payload
// or
setTxPayload 0x00 0x0F // PACKET_LEN = 14; PSDU_LEN = 15; default payload
// or
setTxPayload 0x00 0x11 // PACKET_LEN = 16; PSDU_LEN = 18; longest payload using default TX length config
// or
setTxLength 17
setTxPayload 0x00 0x12 // PACKET_LEN = 18; PSDU_LEN = 19; adjust TX length to prevent underflow error

tx 1
```

For the SubGHz PHYs (2B header), the Frame Length field is the 3 MSB bits of the first byte and all the 8 bits of the second byte in MSB first order.

```
setTxPayload 0x00 <PHR[0:5] + FL[8:11] >> 5> <FL[0:7]>
```

where FL can range between 4 - 2047.

```
CRC_LEN = 2 // default configuration
PHR_LEN = 2

FL = PSDU_LEN = PACKET_LEN
```

More examples:

```
rx 0

config863GHz802154
// or
config915MHz802154

setTxPayload 0x00 0x18 0xA0 // PACKET_LEN = PSDU_LEN = 5; shortest payload
// or
setTxPayload 0x00 0x18 0x70 // PACKET_LEN = PSDU_LEN = 14; default payload
// or
setTxPayload 0x00 0x18 0x08 // PACKET_LEN = PSDU_LEN = 16; longest payload using default TX length config
// or
setTxLength 17
setTxPayload 0x00 0x18 0x88 // PACKET_LEN = PSDU_LEN = 17; adjust TX length to prevent underflow error
```

In summary, to transmit a packet with N bytes of payload (i.e., MHR + MAC payload), set:

- FL to N + CRC_LEN, and
- MAX_PACKET_LEN to N + PHR_LEN, if needed.

The default RX and TX buffer size of RAILtest is set to 512 bytes.

The 863 and 915 MHz inbuilt PHYs limits the payload length to 127 (0xFE).

Use the `printTxPacket` command to see the TX payload!

Use the `configRxOptions 0x1` to store the CRC on valid packet reception.

2. Address Filtering

In this example, we will use the 2.4 GHz PHY only.

Enable Address Filtering

HW accelerated MAC features, such as frame and address filtering, are enabled by enabling 802.15.4 mode using the `enable802154` command.

```
rx 0
config2p4GHz802154
enable802154 rx 100 192 1000
rx 1
```

The default packets transmitted after these commands won't be received on the RX node, because the 802.15.4 mode automatically enables filtering without any address defined (except broadcast messages).

We should format a valid broadcast message to get the packet received without defining any address.

Formatting Broadcast Message

Configure the Frame Control Field as follows. Pay attention to the bit endianness.

- Frame Type = 3 b000 (Beacon)
- Security Enabled = 1b0
- Frame Pending = 1b0
- AR = 1b0
- PAN ID Compression = 1b0
- Reserved = 1b0

The first byte of the PSDU is 0x00 .

- Sequence Number Compression = 1b0
- IE Present = 1b0
- Destination Addressing Mode = 0b10 (Short, 16 bit)
- Frame Version = 1b0
- Source address = 2b10 (Short, 16 bit)

The second byte is 0x88 .

The FCF field is followed by the Sequence Number. This is a random number; we can set it arbitrarily in this example:

- Sequence Number = 0x00

The third byte is 0x00 .

Set the Addressing fields next to the Sequence Number starting by the Destination:

- Destination PAN ID = 0xFFFF (Broadcast)
- Destination Address = 0xFFFF (Broadcast)
- Source PAN ID = 0x1234
- Source Address = 0x5678

The next 8 bytes are 0xFF 0xFF 0xFF 0xFF 0x34 0x12 0x78 0x56 .

And finally, we add 1 byte of MAC payload to the packet:

- Payload data = 0x55

The last byte configured in the TX payload is 0x55.

The total length of the packet is 15 bytes including the header (i.e., the PHR field). Set the first byte according to Example 1.

- FL = PSDU_LEN = 14 = 0x0E (11B MHR + 1B payload + 2B CRC)

```
settxpayload 0 0x0E 0x00 0x88 0x00 0xFF 0xFF 0xFF 0xFF 0x34 0x12 0x78 0x56 0x55
| PHR | FCF | SN | D. PAN | D. Addr. | S. PAN | S. Addr | Payl.
```

This packet is now received without defining the address of the receiver.

Short Address Example

As the next step, we set up an address on the receiver node.

The `setPanId802154` command configures the PAN ID of the device and `setShortAddr802154` sets the Short Address.

To keep the example simple, we set the PAN ID and Address on the RX node to 0x5555 and 0xAAAA, respectively.

```
// TX node

setPanId802154 0x1234
setShortAddr802154 0x5678

// RX node

setPanId802154 0x5555
setShortAddr802154 0xAAAA
```

Now we can send a message directly addressed to our receiver node, keeping all other parameters of the packet the same as before:

```
settxpayload 0 0x0E 0x00 0x88 0x00 0x55 0x55 0xAA 0xAA 0x34 0x12 0x78 0x56 0x55
```

Setting PAN ID and Short Address on the transmitter does not filter packets from transmitting.

You can check at this point that changing only 1 bit in the Destination Address or PAN ID makes the RX node not receive the packet:

```
settxpayload 0 0x0E 0x00 0x88 0x00 0x55 0x57 0xAA 0xAA 0x34 0x12 0x78 0x56 0x55
// or
settxpayload 0 0x0E 0x00 0x88 0x00 0x55 0x55 0xAE 0xAA 0x34 0x12 0x78 0x56 0x55
```

However, RAIL supports multiple addresses and PAN IDs defined at the same time.

You can register a new address by passing a second, optional argument to the `setPanId802154` and `setShortAddr802154` commands.

To receive the previous two wrongly formatted packets, use the following commands:

```
// RX node

setPanId802154 0x5755 1
setShortAddr802154 0xAAAA 1

setPanId802154 0x5555 2
setShortAddr802154 0xAAAE 2
```


Notice that the PAN ID and the address must be defined in pairs on a given index. The following packet still can't be received with the registered PAN ID and Short Address pairs:

```
settxpayload 0 0x0E 0x00 0x88 0x00 0x55 0x57 0xAE 0xAA 0x34 0x12 0x78 0x56 0x55
              | |
```

Long Address Example

As a last address filtering example, receive a packet using the long address.

Set the Source address to `0x0123456789ABCDEF` with the `setLongAddr802154` command.

The TX packet shall be changed:

- Destination address = `2b11` (Long, 64 bit)

The second byte of the FCF is `0x8C`.

- Destination address is now 4 times longer (8 bytes)

The addressing fields (14 bytes) are: `0x55 0x55 0x01 0x23 0x45 0x67 0x89 0xAB 0xCD 0xEF 0x34 0x12 0x78 0x56`.

- We should add 6 bytes to the PHR:
- `FL = PSDU_LEN = 20 = 0x14` (17B MHR + 1B payload + 2B CRC)

```
settxpayload 0 0x14 0x00 0x8C 0x00 0x55 0x55 0x01 0x23 0x45 0x67 0x89 0xAB 0xCD 0xEF 0x34 0x12
              | PHR | FCF | SN | D. PAN | Destination Long Address | S. PAN |
0x78 0x56 0x55
S. Addr | Payl.
```

// or load it in two commands if the default argument length does not let the command pass in one run

```
settxpayload 0 0x14 0x00 0x8C 0x00 0x55 0x55 0x01 0x23 0x45 0x67
settxpayload 10 0x89 0xAB 0xCD 0xEF 0x34 0x12 0x78 0x56 0x55
```

Don't forget to increase the maximum packet length on the transmitter if needed:

```
setTxLength 19
```

As a complete example to send and receive a packet with a long address, use the following commands after resetting the devices:

```
// RX Node

rx 0
config2p4GHz802154
enable802154 rx 100 192 1000
rx 1
setPanId802154 0x5555
setLongAddr802154 0x01 0x23 0x45 0x67 0x89 0xAB 0xCD 0xEF

// TX Node

rx 0
config2p4GHz802154
enable802154 rx 100 192 1000
rx 1
settxpayload 0 0x14 0x00 0x8C 0x00 0x55 0x55 0x01 0x23 0x45 0x67
settxpayload 10 0x89 0xAB 0xCD 0xEF 0x34 0x12 0x78 0x56 0x55
settxlength 19
```

In the examples above, the transmitted frame was a Beacon type message. By default, enabling 802.15.4 mode, all frame types are enabled to receive. To filter out or reenables messages with different frame types, use the `acceptFrames` command.

The command's argument sets which frames can be received:

```
acceptFrames <MAC command> <ACK> <Data> <Beacon> [<Multipurpose>]
```

For example, after the following command, the receiver will reject data and Beacon type frames, but accept Ack and MAC command frames.

```
acceptFrames 1 1 0 0
```

Notes on Address Filtering

Broadcast messages can always be received regardless of the configured addresses.

Addresses can be deactivated by setting the address to the broadcast address (0xFFFF) at the corresponding index.

```
setPanId802154 0xFFFF <index>
setShortAddr802154 0xFFFF <index>
```

802.15.4 Address Filtering is fully disabled when the device is in promiscuous mode.

```
setPromiscuousMode 1
```

There is no constraint between the short and the long address and the PAN ID; you can set each of them independently at any index.

RAIL supports intra-PAN communication via multiple addresses defined using different PAN IDs.

Auto-ACK

Auto-ACK is automatically enabled by enabling 802.15.4 mode using the `enable802154` command.

```
rx 0
config2p4GHz802154
enable802154 rx 100 192 1000
rx 1
```

The third and the fourth parameter of the `enable802154` command sets the RX-to-TX turnaround time and the ACK timeout in microseconds, respectively.

Let's start with the TX packet and corresponding device configuration introduced in the previous example:

```
settxpayload 0 0x0E 0x00 0x88 0x00 0x55 0x55 0xAA 0xAA 0x34 0x12 0x78 0x56 0x55
```

To request an Acknowledge message, the first byte of the FCF field must be changed:

1. Request ACK packet by setting the AR bit (5th bit).
2. Beacon frames cannot be acknowledged: set Frame type (3 LS bits) to `3b001` (Data).

The first byte of the FCF shall be `0x21`:

```
settxpayload 0 0x0E 0x21 0x88 0x00 0x55 0x55 0xAA 0xAA 0x34 0x12 0x78 0x56 0x55
```

Also, RAIL must be informed that, after transmitting the packet, an ACK packet is expected to be received. To do that, issue the following command:

```
configTxOptions 1
```

See a complete example for testing Auto-ACK feature below:

```
// TX node

rx 0
config2p4GHz802154
enable802154 rx 100 192 1000
setPanId802154 0x1234
setShortAddr802154 0x5678
setTxPayload 0 0x0E 0x21 0x88 0x00 0x55 0x55 0xAA 0xAA 0x34 0x12 0x78 0x56 0x55
configTxOptions 1
rx 1

// RX node

rx 0
config2p4GHz802154
enable802154 rx 100 192 1000
setPanId802154 0x5555
setShortAddr802154 0xAAAA
rx 1
```

Review the received ACK message's payload:

```
0x05 0x02 0x00 0x00
```

- `0x05` : This is the PHR and the Frame Length is 5
- `0x02 0x00` : the FCF field indicates that it is an ACK frame
- `0x00` : Sequence Number is identical to the packet which is acknowledged by this packet

To change the Sequence Number only, you can use the following command:

```
setTxPayload 3 0x01
```

802.15.4 ACK frames are automatically assembled in RAIL, there is no need to set it up in the RX node.

Notes on Auto-ACK

Auto-ACK is fully disabled when the device is in promiscuous mode.

```
setPromiscuousMode 1
```

To disable or reenables auto ACK feature use the following command:

```
autoAckPause <RX> <TX>
```

Changing the ACK payload or its length prevents RAIL to assemble the packet. Also, printing the ACK packet will not result in printing the auto 15.4 ACK packet.

```
setAckPayload
setAckLength
printAckPacket
```

RAIL sets the Frame Pending bit in the ACK payload automatically.

Runtime PHY changes

Whitening and CRC Config

The SUN FSK PHYs define dynamic radio config changes (i.e., selecting 2/4 byte CRC - FCS type - or enabling/disabling whitening - DW - over the PSDU field) which is fully supported by RAIL. The radio is reconfigured on the fly during packet transmission and reception according to the PHR field.

This mode can be enabled by the `RAIL_IEEE802154_G_OPTION_GB868` option. Use the `set802154g` command to enable the `G_OPTION`s.

```
// TX node

rx 0
config915MHz802154
enable802154 rx 100 192 1000
set802154g 1
setPanId802154 0x1234
setShortAddr802154 0x5678
settxpayload 2 0x00 0x88 0x00 0x55 0x55 0xAA 0xAA 0x34 0x12 0x78 0x56 0x55
rx 1

settxpayload 0x00 0x00 0x70 // not-whitened & 4B FCS
settxpayload 0x00 0x08 0x70 // not-whitened & 42 FCS
settxpayload 0x00 0x10 0x70 // whitened & 4B FCS
settxpayload 0x00 0x18 0x70 // whitened & 2B FCS

// RX node

rx 0
config915MHz802154
enable802154 rx 100 192 1000
set802154g 1
setPanId802154 0x5555
setShortAddr802154 0xAAAA
rx 1
```

Alternatively, you can use the `set802154PHR` command once the radio is configured with a PHY capable of handling dynamic PHY reconfiguration.

For SUN FSK compatibility, the first argument should be 1. The second and third arguments set up which CRC is being used and whether whitening is enabled on the transmitted packet.

This command also sets up the frame length derived from the `setTxLength <PACKET_LEN>` config.

For the 2.4 GHz PHY, pass 0 as the first and omit the other arguments to set up the Frame Length in the PHR according to the configured TX packet length.

```
settxpayload 0x00 0x00 0x70 // not-whitened & 4B FCS
set802154PHR 1 0 0

settxpayload 0x00 0x08 0x70 // not-whitened & 42 FCS
set802154PHR 1 1 0

settxpayload 0x00 0x10 0x70 // whitened & 4B FCS
set802154PHR 1 0 1

settxpayload 0x00 0x18 0x70 // whitened & 2B FCS
set802154PHR 1 1 1
```

Dynamic FEC

SUN FSK protocol defines the dynamic FEC feature which is supported on certain EFR32 device generations. The SFD field type indicates whether FEC is enabled or not on the transmitted/received packet.

This mode can be enabled by the `RAIL_IEEE802154_G_OPTION_DYNFEC` option. Use the `set802154g` command to enable the `G_OPTION` s.

```
// TX node

rx 0
enable802154 rx 100 192 1000

//set_appropriate_channel
setconfigindex <Y>
setchannel <X>

set802154g 3
setPanId802154 0x1234
setShortAddr802154 0x5678
settxpayload 2 0x00 0x88 0x00 0x55 0x55 0xAA 0xAA 0x34 0x12 0x78 0x56 0x55
settxpayload 0x00 0x18 0x70 // whitened & 2B FCS
rx 1

// RX node

rx 0
setchannel <set_appropriate_channel>
enable802154 rx 100 192 1000
set802154g 3
setPanId802154 0x5555
setShortAddr802154 0xAAAA
rx 1
```

Transmit packet without FEC encoding (using syncword #0):

```
configTxOptions 0
tx 1
```

Transmit packet without FEC encoding (using syncword #1):

```
configTxOptions 0x4
tx 1
```

Using it on platforms where it is supported requires help from technical support to create an appropriate PHY.

The `set802154g` command requires an appropriate PHY configuration and idle state.

Power Amplifier Power Conversion Functions in RAIL 2.x

Power Amplifier Power Conversion Functions in RAIL 2.x

This section replaces *AN1127: Power Amplifier Power Conversion Functions in RAIL 2.x*. Further updates to this application note will be provided here.

In many wireless applications, due to limitations imposed by the transmission protocol, law, or energy constraints, it is critical to know the power output of a given device. However, this issue is complicated by the fact that output characteristics may vary drastically even for a given chip, based on the board or module onto which the chip is mounted, due to poor impedance matching, or due to parasitics. This application note outlines how to account for the variation across custom boards and applications for the Silicon Labs EFR32 family of chips. While the Proprietary Flex SDK is required for the procedures described in this document, the resulting calibration curves can be used with applications developed with Silicon Labs Connect, Zigbee, Bluetooth, and OpenThread SDKs. Customers developing with Z-Wave should refer to https://www.silabs.com/community/wireless/z-wave/knowledge-base.entry.html/2020/03/03/z-wave_700_tx_powercalibrationandadjustment-YUiB. Proprietary is supported on all EFR32FG devices. For others, check the device's data sheet under Ordering Information > Protocol Stack to see if Proprietary is supported. In Proprietary SDK version 2.7.n, Connect is not supported on EFR32xG22.

EFR32 Power Amplifiers And RAIL

Overview: EFR32 Power Amplifiers and RAIL

The EFR32 families of chips each come equipped with three Power Amplifiers (PAs):

- EFR32xG1x
 - A high-power * 2.4 GHz PA (for power 20 dBm and lower)**
 - A low-power 2.4 GHz PA (for power 0 dBm and lower)
 - A Sub-GHz PA**
- EFR32xG21
 - A high-power 2.4 GHz PA (for power 20 dBm and lower)**
 - A medium-power 2.4 GHz PA (for power 10 dBm and lower)
 - A low-power 2.4 GHz PA (for power 0 dBm and lower)
- EFR32xG22
 - A high-power 2.4 GHz PA (for power 6 dBm and lower)**
 - A low-power 2.4 GHz PA (for power 0 dBm and lower)
- EFR32xG23x02x
 - A high-power Sub-GHz PA (for power 20 dBm and lower)**
- EFR32xG23x01x
 - A high-power Sub-GHz PA (for power 14 dBm and lower)**
- EFR32xG24x02x
 - A high-power 2.4 GHz PA (for power 20 dBm and lower)**
- EFR32xG24x01x
 - A high-power 2.4 GHz PA (for power 10 dBm and lower)**
 - A low-power 2.4 GHz PA (for power 0 dBm and lower)
- EFR32xG25
 - A single PA for OFDM operation
 - A single PA for SUBGIG (FSK) operation
- EFR32xG26x02x
 - A high-power 2.4 GHz PA (for power 20 dBm and lower)**
- EFR32xG26x01x
 - A high-power 2.4 GHz PA (for power 10 dBm and lower)**
 - A low-power 2.4 GHz PA (for power 0 dBm and lower)
- EFR32xG27x02x
 - A high-power 2.4 GHz PA (for power 4 dBm and lower)**
 - A low-power 2.4 GHz PA (for power 0 dBm and lower)
- EFR32xG27x01x
 - A high-power 2.4 GHz PA (for power 6/8 dBm and lower)**
 - A low-power 2.4 GHz PA (for power 0 dBm and lower)
- EFR32xG28x02x
 - A high-power 2.4 GHz PA (for power 20 dBm and lower)**
 - A high-power Sub-GHz PA (for power 20 dBm and lower)**
- EFR32xG28x01x
 - A high-power 2.4 GHz PA (for power 14 dBm and lower)**
 - A high-power Sub-GHz PA (for power 14 dBm and lower)**

Each of these PAs has a unique number of discrete "power levels", which are simply abstractions of the different register settings that control the active PA. For each PA, the following power levels are available:

- EFR32xG1x
 - High-power 2.4 GHz: 0-252**
 - Low-power 2.4 GHz: 1-7

- Sub-GHz: 0-248
- EFR32xG21
 - High-power 2.4 GHz: 1-180**
 - Medium-power 2.4 GHz: 1-90
 - Low-power 2.4 GHz: 1-64
- EFR32xG22
 - High-power 2.4 GHz: 0-127**
 - Low-power 2.4 GHz: 0-15
- EFR32xG23x02x
 - High-power Sub-GHz: 0-240**
- EFR32xG23x01x
 - High-power Sub-GHz: 0-240**
- EFR32xG24x02x
 - High-power 2.4 GHz: 0-180**
- EFR32xG24x01x
 - High-power 2.4 GHz: 0-90**
 - Low-power 2.4 GHz: 0-15
- EFR32xG25
 - "power levels" not supported on dBm-to-powerSetting Pas (see 2.3 dBm-to-powerSetting Mode)
- EFR32xG26x02x
 - High-power 2.4 GHz: 0-180**
- EFR32xG26x01x
 - High-power 2.4 GHz: 0-90**
 - Low-power 2.4 GHz: 0-15
- EFR32xG27x02x
 - High-power 2.4 GHz: 0-127**
 - Low-power 2.4 GHz: 0-15
- EFR32xG27x01x
 - High-power 2.4 GHz: 0-127**
 - Low-power 2.4 GHz: 0-15
- EFR32xG28x02x
 - High-power 2.4 GHz: 0-240**
 - High-power Sub-GHz: 0-240**
- EFR32xG28x01x
 - High-power 2.4 GHz: 0-240**
 - High-power Sub-GHz: 0-240**

*Note that the use of 'high', 'medium', and 'low' in the names of these PAs refers to power consumption, not power output. It is possible, for instance, to configure the high-power PA to transmit at a lower dBm output than the low-power PA.

**Maximum power/use of these PAs may be restricted by your specific OPN. Please see the data sheet for more details regarding your particular part.

Although the number of power levels vary across PAs, for a given PA a higher power level generally indicates a higher dBm output power. It is important to realize, however, that a PA power level does not always correspond to the same dBm output power across boards and applications.

The power level of the active PA is controlled by the `RAIL_GetTxPower` and `RAIL_SetTxPower` APIs. `RAIL_GetTxPowerDbm` and `RAIL_SetTxPowerDbm` also allow users to interface with dBm output power values, but they are merely utility functions that combine calls to `RAIL_ConvertRawToDbm`, `RAIL_ConvertDbmToRaw`, `RAIL_SetTxPower`, and `RAIL_GetTxPower`. They add no new functionality or features on their own. See the RAIL documentation that comes with the Simplicity Studio SDK for more details.

Although these PA power levels add another layer of complexity to PA control, they give the user much more control to account for parasitics and impedance mismatch between the PA and a custom board. In RAIL 1.x, customers were only able to interface with PA output power in dBm and provide a linear offset, which was often not sufficiently robust to account for the complex PCB-to-PA interaction. Now, RAIL allows users to assign an exact dBm power to each of the PA power levels for their custom boards through the functions `RAIL_ConvertDbmToRaw` and `RAIL_ConvertRawToDbm`. Note that these functions convert between raw power levels and deci-dBm, or dBm * 10, to allow for higher precision dBm values. Though Silicon Labs provides a default implementation for these functions to do the conversion, the user can also override these functions with other algorithms, ranging from a highly precise but space-intensive lookup table for all possible PA-power

level combinations, to less robust solutions such as a lookup table containing only a few power levels that correspond to the dBm output power levels of interest (see [Alternative Methods to Convert between Power Levels and dBm Output Power](#) for more information on creating custom conversion functions).

In the default conversion function implementation, Silicon Labs uses an 8-segment, piecewise linear curve fit of the mapping between PA power levels and dBm output powers to convert between these quantities. On EFR32xG1x chips, there is one curve for the high- power 2.4GHz PA, and another for the sub-GHz PA. On EFR32xG21 and EFR32xG23 chips, there is one curve for each PA. On EFR32xG22 and EFR32xG24 chips, there is one curve for the high power 2.4GHz PA. For these PAs, this method provides a good balance between accuracy across power levels, computational speed, and code size. The conversion for low-power 2.4GHz on EFR32xG1x, EFR32xG22, and EFR32xG24, which have fewer than 20 possible values, is handled with a simple lookup table. The majority of this application note focuses on how to generate the curve data (slopes and y-intercepts) for a custom application, similar to the default values currently found in `pa_curves_efr32xg1x.h` and `pa_curves_efr32xg2x.h` in RAIL 2.8 and earlier versions and in `sl_rail_util_pa_curves.h` in RAIL 2.9 and later versions. In other words, it assumes that the existing methodologies found in `RAIL_ConvertDbmToRaw` and `RAIL_ConvertRawToDbm` are appropriate for your application and will not be over- written, but that the parameters for these functions need to be adjusted for the custom board.

This application note is not intended for use with Silicon Labs' pre-certified Wireless SiP and PCB modules. The power amplifiers on these parts have been calibrated specifically to adhere to certain regulations. Trying to alter these calibrations in software will cause the RAIL library to assert and lock the chip, preventing further use until Silicon Labs' default libraries are restored.

In the following procedures, use the `railtest` and `pa_customer_curve_fits.py` script from RAIL 2.5 or higher in Proprietary Flex SDK version 2.4.0 or higher, even if you are developing in an earlier Proprietary Flex SDK 2.x version. There were improvements to the app and script that will subtly improve results. If you have already generated the CSV files from the procedure in [Piecewise Linear Curve Fits](#), you can simply re-run the newer `pa_customer_curve_fits.py` on the existing CSVs. There is no need to re-generate the CSVs by running the sweep again.

Generating Conversion Data

Generating Conversion Data

Piecewise Linear Curve Fits

The following steps explain how to generate the piecewise fit curves for the following PAs:

- EFR32xG1x: High-power 2.4 GHz and sub-GHz
- EFR32xG21: All PAs
- EFR32xG22: High-power 2.4 GHz
- EFR32xG23: All PAs
- EFR32xG24: High-power 2.4 GHz

The procedure uses the 2.4 GHz high-power PA on EFR32xG1x devices as an example, but the same procedure is applicable to all the aforementioned PAs. For the other PAs, which do not use the piecewise linear fit approach, see the Lookup Table Fits section below.

1. In Simplicity Studio, generate a RAILtest application configured with the desired PA, PA power source, and PA ramp time. The configuration process depends on the Proprietary Flex SDK version. If you are using version 2.x with Simplicity Studio 4, use Hardware Configurator (see *AN1115: Configuring Peripherals for 32-Bit Devices using Hardware Configurator* for more detail). If you are using version 3.x with Simplicity Studio 5, use the Component Editor to modify the **RAIL Utility, Power Amplifier** component. Build the RAILtest application, and flash the resulting binary to the chip mounted to the PCB you are trying to characterize.
2. Connect the RF output of the board (U.FL test connector, SMA connector, or similar, with antenna disconnected from the RF path) to the input of a spectrum analyzer. The spectrum analyzer should be in dBm peak detection mode to sense values between approximately -30 to 20 dBm and centered on the base frequency/channel for your PHY and channel combination. Make sure to account for any cable / adapter loss in your measurements.
3. Connect to the RAILtest CLI and enter the command `sweepTxPower`. For more information on RAILtest functionality and the CLI interface, see *Using RAILTest* in the Simplicity Studio Flex SDK RAIL documents group (RailTestUserGuide.html).
4. Your chip should begin a CW tone at power level 1. Record the power level and dBm power measured on the spectrum analyzer in a CSV file in the format `<POWER_LEVEL>,<ACTUAL_DBM_OUTPUT>` with one reading per line. Note that unlike the conversion functions `RAIL_GetTxPowerDbm` and `RAIL_SetTxPowerDbm`, which use deci-dBm values, values in these csv files should be specified in dBm. Sample files **SubgigPowerMapping.csv** and **2p4PowerMapping.csv** are included in the SDK for reference, and can be found in the `/platform/radio/rail_lib/chip/efr32/efr32xg1x/characterization` folder of the SDK installation.
5. Repeat step 4 for all power levels. Press Enter in the RAILtest CLI to progress to the next power level.
6. From a terminal prompt, run the `pa_customer_curve_fits.py` script (in `/platform/radio/rail_lib/tools`), with the `python pa_customer_curve_fits.py <CSV_FILE>`. If you don't already have them installed, you will need to install the **numpy** and **pylab** python libraries. Running the commands `pip install numpy` and `pip install matplotlib` will install those libraries in the appropriate directories on your machine to be accessible by python. The following figure shows an example of an intended result, after the `pa_customer_curve_fits.py` script has been run on the **2p4PowerMapping.csv** file from a terminal prompt.

Note: The script also provides extra parameters such as `-maxPower/-m` and `-increment/-i` to accommodate the maximum power from the csv and to divide the data in segments of needed "increment" dBm steps.

```
> python tools/pa_customer_curve_fits.py chip/efr32/efr32xg1x
/characterization/2p4PowerMapping.csv

RAIL_TxPowerCurveSegment_t[] C Structure
{ { 252, 2776, -300026 }, \
  { 136, 1335, -73192 }, \
  { 83, 772, -7179 }, \
  { 50, 441, 17309 }, \
  { 41, 343, 22520 }, \
  { 22, 209, 22360 }, \
  { 14, 124, 18896 }, \
  { 8, 75, 14632 } }
```

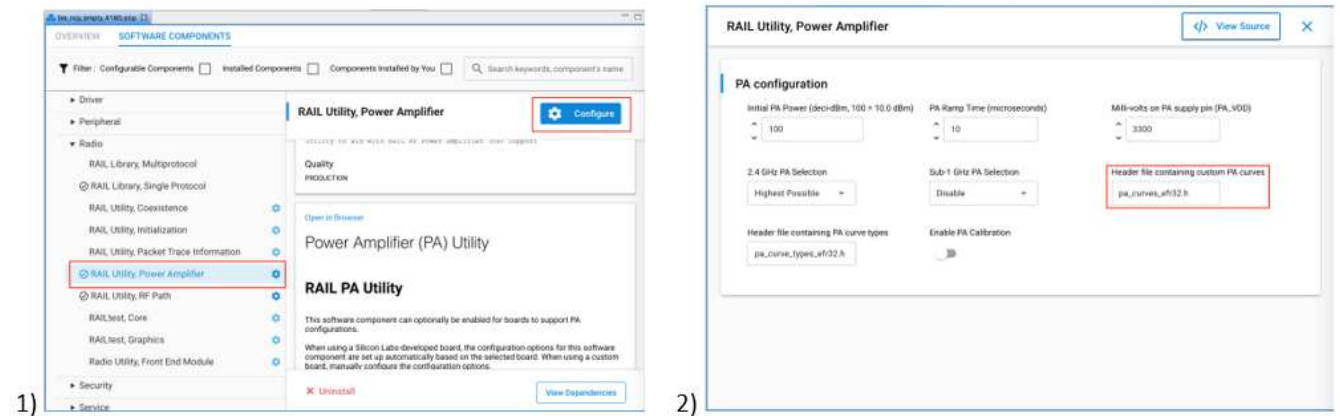
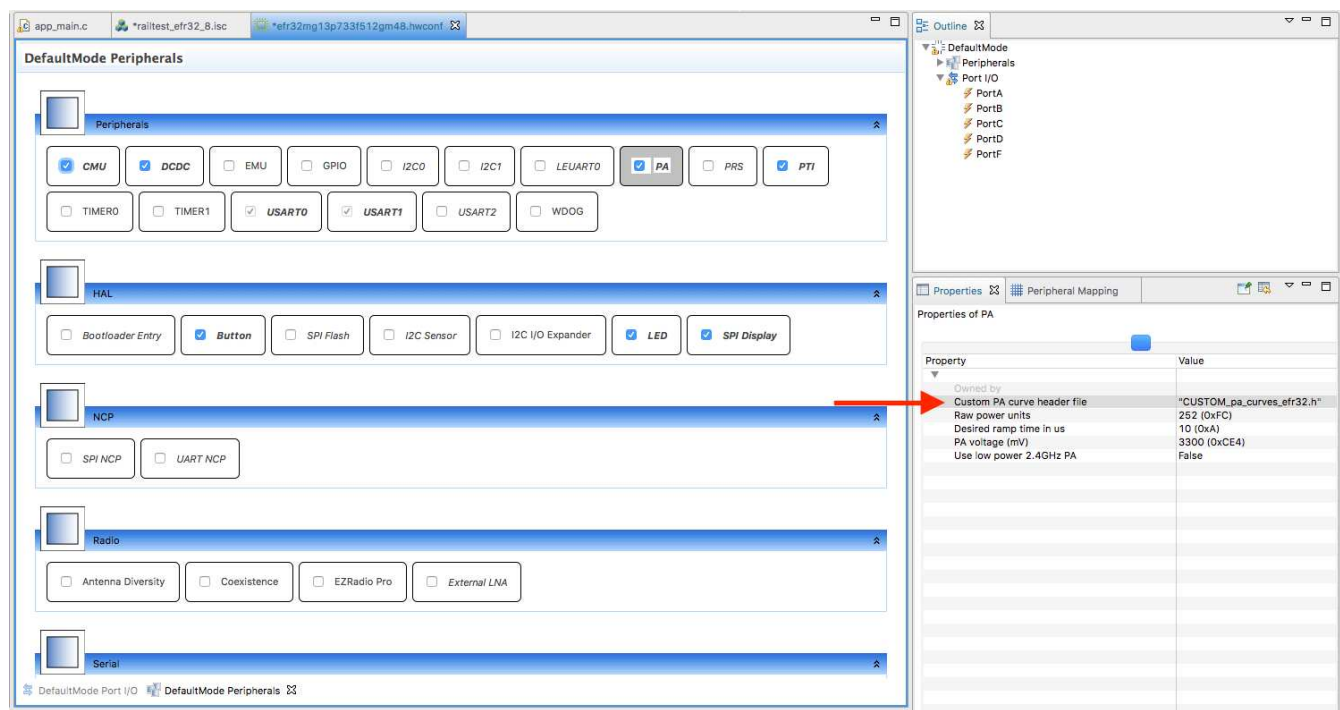
7. If using multiple PAs (other than the EFR32xG1x 2.4 GHz low-power PA), repeat steps 1-6 for the PA you did not characterize previously.
8. Create a copy of the PA curve file (here and in subsequent steps, "PA curve file" refers to **pa_curves_efr32xg1x.h** or **pa_curves_efr32xg2x.h** in RAIL 2.8 and earlier versions, and **sl_rail_util_pa_curves.h** in RAIL 2.9 versions and later depending on the platform currently being characterized). Copy the terminal output from the python script under the appropriate macro. Note that there is one `RAIL_PA_CURVES_...` define for each combination of PA and PA power source available for your hardware. You do not need to copy data for a PA/PA power supply (`...VBAT...` or `...DCDC...`) combination that is not being used. Additionally, update the `...MIN_POWER` and `...MAX_POWER` macros to reflect the values you observed. The following figure shows an example of the copying process, where the values from the previous figure have been copied into the appropriate (Sub-GHz and 2.4 GHz, battery-powered) macros in a new **CUSTOM_pa_curves_efr32xg1x.h** file. Some of the `...MIN_POWER` and `...MAX_POWER` macros have been updated.

```

36 #ifndef __PA_CURVES_EFR32_H_
37 #define __PA_CURVES_EFR32_H_
38
39 #define RAIL_PA_CURVES_LP_VALUES 7
40 #define RAIL_PA_CURVES_2P4_HP_SG_PIECEWISE_SEGMENTS 8
41
42 #define RAIL_PA_CURVES_2P4_HP_VBAT_MAX_POWER 180
43 #define RAIL_PA_CURVES_2P4_HP_VBAT_MIN_POWER -280
44 #define RAIL_PA_CURVES_2P4_HP_VBAT_CURVES \
45 { { 248, 2761, -301724 }, \
46   { 157, 1359, -81052 }, \
47   { 116, 760, -11171 }, \
48   { 73, 416, 14056 }, \
49   { 51, 329, 18564 }, \
50   { 18, 209, 18160 }, \
51   { 8, 116, 14132 }, \
52   { 4, 75, 10593 } }
53
54 #define RAIL_PA_CURVES_2P4_HP_DCDC_MAX_POWER 140
55 #define RAIL_PA_CURVES_2P4_HP_DCDC_MIN_POWER -260
56 #define RAIL_PA_CURVES_2P4_HP_DCDC_CURVES \
57 { { -1, 0, 0 }, \
58   { 252, 4306, -391604 }, \
59   { 129, 1435, -52495 }, \
60   { 80, 610, 13579 }, \
61   { 38, 331, 24456 }, \
62   { 24, 224, 23902 }, \
63   { 14, 140, 20330 }, \
64   { 8, 81, 15607 } }
65
66 #define RAIL_PA_CURVES_SG_VBAT_MAX_POWER 200
67 #define RAIL_PA_CURVES_SG_VBAT_MIN_POWER -260
68 #define RAIL_PA_CURVES_SG_VBAT_CURVES \
69 { { 248, 2761, -301724 }, \
70   { 157, 1359, -81052 }, \
71   { 116, 760, -11171 }, \
72   { 73, 416, 14056 }, \
73   { 51, 329, 18564 }, \
74   { 18, 209, 18160 }, \
75   { 8, 116, 14132 }, \
76   { 4, 75, 10593 } }

```

9. Update the hardware config `HAL_PA_CURVE_HEADER` header to point to your new file. The following figures illustrates where to update the header. The interface depends on the tool used (Hardware Configurator in Proprietary Flex SDK 2.x, or Component Editor in Proprietary Flex SDK 3.x).



10. For customers using only one or two of the available PAs, Silicon Labs recommends removing references to these functions from `pa_conversions_efr32.c` and from the `RAIL_DECLARE_TX_POWER` macros in your new version of the PA curves file. Although not strictly necessarily, doing this can save substantial code size versus compiling in useless data for a PA that is never engaged.

Lookup Table Fits

This information applies to the following PAs:

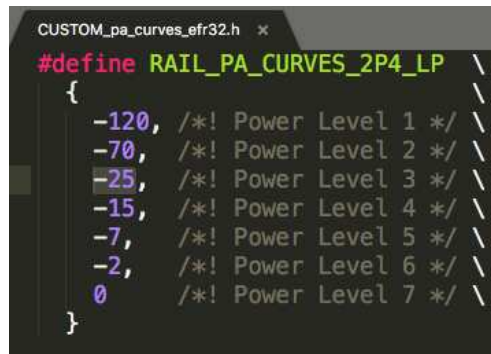
- EFR32xG1x: Low-power 2.4 GHz
- EFR32xG22: Low power 2.4 GHz
- EFR32xG24x01x: Low power 2.4 GHz

With only a few unique power levels, some PAs on some chip families use a simple lookup table to translate power levels to output powers, as opposed to a more code-size-intensive curve fit. Using the EFR32xG1x low-power PA as an example, follow steps 1-5 in the previous section, but for step 4, place the results directly into the `RAIL_PA_CURVES_2P4_LP` macro of the custom PA curves file. These values should be entered as deci-dBm (for example, enter -100 to indicate -10 dBm). A similar process can be followed for any other PAs using the lookup table approach. The following two figures contain an example of this process.

```
sweepTxPower{{{sweepTxPower}}}{Sweeping:Started}{Instructions:'q' to quit or 'enter' to continue.}}
{{{sweepTxPower}}}{PowerLevel:1}}

{{{sweepTxPower}}}{PowerLevel:2}}

{{{sweepTxPower}}}{PowerLevel:3}}
```



```
CUSTOM_pa_curves_efr32.h
#define RAIL_PA_CURVES_2P4_LP \
{ \
    -120, /*! Power Level 1 */ \
    -70, /*! Power Level 2 */ \
    -25, /*! Power Level 3 */ \
    -15, /*! Power Level 4 */ \
    -7, /*! Power Level 5 */ \
    -2, /*! Power Level 6 */ \
    0, /*! Power Level 7 */ \
}
```

dBm-to-powerSetting Mode

A new mode has been added to the EFR32xG25 family that allows for a single PA to support the entire range of supported dBm levels for both the OFDM and SUBGIG (FSK) PAs. This is accomplished with a table in memory that contains a "powerSetting" for each supported dBm level. The table specifies the minimum and maximum dBm levels that are supported, as well as the step size between entries in the table. The default step size is 0.1 dBm. This allows for precise control over the output power of the PA, making it possible to achieve the desired level of performance across the entire range of supported dBm levels. Note that this new mode utilizes "powerSetting" instead of the traditional "power levels" for the output power of the PA. As a result, the capability to set a specific "raw" power level is no longer supported in this mode. Only the setting of dBm levels from the table is supported in this mode.

OFDM

A powerSetting for OFDM is comprised of three primary components: the digital gain, the slice level, and the filter gain. The digital gain is specified by the VPA_OFDM_GAINDIG_MASK and VPA_OFDM_GAINDIG_SHIFT values, which define a mask and shift for extracting the digital gain from the powerSetting value. The slice level is specified by the VPA_OFDM_SLICE_MASK and VPA_OFDM_SLICE_SHIFT values, which define a mask and shift for extracting the slice level from the powerSetting value. Finally, the filter gain is specified by the VPA_OFDM_FILGAIN_MASK and VPA_OFDM_FILGAIN_SHIFT values, which define a mask and shift for extracting the filter gain from the powerSetting value. These three components work together to determine the overall power output of the PA for OFDM signals.

To ensure proper usage and optimal performance, it is important to adhere to the valid ranges for the digital gain, slice level, and filter gain. The valid range for the digital gain is 0 to 204 (the value is multiplied by 5 in RAIL, resulting in a range of 0 to 1020), the valid range for the slice level is 0 to 191, and the valid range for the filter gain is 0 to 3.

In addition to the valid ranges, the following are the recommended ranges: digital gain between 200 and 800, filter gain of 2.

```
#define VPA_OFDM_GAINDIG_MASK    (0xFFUL)
#define VPA_OFDM_GAINDIG_SHIFT  (0U)
#define VPA_OFDM_SLICE_MASK     (0xFF00UL)
#define VPA_OFDM_SLICE_SHIFT    (8U)
#define VPA_OFDM_FILGAIN_MASK   (0xFF0000UL)
#define VPA_OFDM_FILGAIN_SHIFT  (16U)
```

To generate a table using the `pa_dbm_mapping_table_generator.py` script (in `platform/radio/rail_lib/tools`), you will need a CSV file containing the necessary data for OFDM mode. The file should have a header row with column names for:

- Digital gain
- Slice level

Filter gain

Each subsequent row should contain data for a specific dBm level within the desired range. Each column's data should be separated by a comma, and you can add optional comments for each row using the "#" symbol. It is crucial to ensure that the CSV file is properly formatted, with the correct data in each column and the correct number of rows for the desired dBm range.

Table: Format of the OFDM PA CSV File

TXTRIMPASLICE	GAINDIG	FILGAIN	#dBm
x	x	x	#-317
x	x	x	#-316
x	x	x	#-315
x	x	x	#-314

This table shows an example of how the CSV file should be formatted for generating a table for OFDM mode. The first row is the header row, which specifies the column names. Each subsequent row contains data on the power settings for a specific dBm level, as well as an optional comment indicating the corresponding dBm value. The data for each column is separated by a comma. It is important to include a row for every dBm level within the desired range, with the appropriate power setting values for each level. Please be aware that the "dBm" column in the table is preceded by a "#" symbol, which indicates that it is a comment and will be ignored by the script.

SUBGIG (FSK)

A powerSetting for SUBGIG (FSK) mode is comprised of four primary components: the fine current setting, the coarse current setting, the slice level, and the stripe setting. The fine current setting is specified by the VPA_SUBGIG_CURRENTFINE_MASK and VPA_SUBGIG_CURRENTFINE_SHIFT values, which define a mask and shift for extracting the fine current setting from the powerSetting value. The coarse current setting is specified by the VPA_SUBGIG_CURRENTCOARSE_MASK and VPA_SUBGIG_CURRENTCOARSE_SHIFT values, which define a mask and shift for extracting the coarse current setting from the powerSetting value. The slice level is specified by the VPA_SUBGIG_SLICE_MASK and VPA_SUBGIG_SLICE_SHIFT values, which define a mask and shift for extracting the slice level from the powerSetting value. Finally, the stripe setting is specified by the VPA_SUBGIG_STRIPE_MASK and VPA_SUBGIG_STRIPE_SHIFT values, which define a mask and shift for extracting the stripe setting from the powerSetting value. These four components work together to determine the overall power output of the PA for SUBGIG (FSK) signals.

The valid range for the fine current setting is 0 to 8, the valid range for the coarse current setting is 0 to 31, the valid range for the slice level is 0 to 2, and the valid range for the stripe setting is 0 to 31.

In addition to the valid ranges, the following are the recommended ranges: current fine of 4, and slice level between 1 and 2.

```
#define VPA_SUBGIG_CURRENTFINE_MASK    (0xFF000000UL)
#define VPA_SUBGIG_CURRENTFINE_SHIFT  (24U)
#define VPA_SUBGIG_CURRENTCOARSE_MASK (0xFF0000UL)
#define VPA_SUBGIG_CURRENTCOARSE_SHIFT (16U)
#define VPA_SUBGIG_SLICE_MASK         (0xFF00UL)
#define VPA_SUBGIG_SLICE_SHIFT        (8U)
#define VPA_SUBGIG_STRIPE_MASK        (0xFFUL)
#define VPA_SUBGIG_STRIPE_SHIFT       (0U)
```

To generate a table using the pa_dbm_mapping_table_generator.py script (in platform/radio/rail_lib/tools), you will need a CSV file containing the necessary data for SUBGIG (FSK) mode. The file should have a header row with column names for:

- Coarse current setting
- Fine current setting
- Slice level
- Stripe setting

Each subsequent row should contain data for a specific dBm level within the desired range. Each column's data should be separated by a comma, and you can add optional comments for each row using the "#" symbol. It is crucial to ensure that

the CSV file is properly formatted, with the correct data in each column and the correct number of rows for the desired dBm range.

Table: Format of the SUBGIG (FSK) PA CSV File

COARSE	FINE	SLICE	STRIPE	#dBm
x	x	x	x	#-317
x	x	x	x	#-316
x	x	x	x	#-315
x	x	x	x	#-314

This table shows an example of how the CSV file should be formatted for generating a table for SUBGIG (FSK) mode. The first row is the header row, which specifies the column names. Each subsequent row contains data on the power settings for a specific dBm level, as well as an optional comment indicating the corresponding dBm value. The data for each column is separated by a comma. It is important to include a row for every dBm level within the desired range, with the appropriate power setting values for each level. Please be aware that the "dBm" column in the table is preceded by a "#" symbol, which indicates that it is a comment and will be ignored by the script.

Generating the Table

The script `pa_dbm_mapping_table_generator.py` is located in the `platform/radio/rail_lib/tools` directory. It takes data from the CSV file specified with the `File` argument and generates a list of power setting values based on the `-min` or `--minPower` and `-max` or `--maxPower` arguments. To understand the CSV format for each respective mode, refer to the OFDM and SUBGIG sections above. Note that a `-t` or `--type` argument is also required, with a default value of `OFDM`, to specify the type of curve.

It also accepts command-line arguments for:

- An optional flag `-f` or `--fem` to indicate whether the table is for a front-end module (FEM) with a default value of `False`.
- An optional argument `-o` or `--output` to specify the output file for the generated code.

The script has a default increment value for the dBm levels, which you can customize as needed. Additionally, the script generates C code for defining the table, including:

- The number of values
- The step size between dBm levels
- The minimum and maximum dBm levels
- The actual power setting values

Finally, the script outputs the generated C code to `stdout`. To use the generated table, the user should replace the applicable contents of the `plugin/pa-conversions/efr32xg25/` files with the output of the script.

Alternative Methods To Convert Between Power Levels And DBm Output Power

Alternative Methods to Convert between Power Levels and dBm Output Power

For your application, having highly granular dBm output levels may not be necessary. Therefore, it may make sense to overwrite the `RAIL_ConvertRawToDbm` and `RAIL_ConvertDbmToRaw` functions that Silicon Labs provides. This example demonstrates how to simplify those functions to return only a few values of interest. Doing this would also allow you to remove calls to the `RAIL_InitTxPowerCurves` function and `RAIL_DECLARE_TX_POWER_XXXX_CURVES` macro. Removing these dependencies on the piecewise linear curve fits allows you to reduce application code size significantly. In this example, the functions are meant to work with common dBm values, such as 20, 13, 3, 0 and -10 (200, 130, 30, 0 and -100 deci-dBm, respectively).

```

RAIL_TxPowerLevel_t RAIL_ConvertDbmToRaw(RAIL_Handle_t railHandle,
    RAIL_TxPowerMode_t mode,
    RAIL_TxPower_t power)
{
    // Anything above this power is likely still an error, so it
    // should return an invalid value
    if (power >= RAIL_TX_POWER_MAX) {
        return RAIL_TX_POWER_LEVEL_INVALID;
    }
    if (mode == RAIL_TX_POWER_MODE_2P4_LP) {
        if (power >= 0) {
            return RAIL_TX_POWER_LEVEL_LP_MAX;
        } else if (power >= -100) {
            return 2;
        } else {
            return 1;
        }
    }
    if (mode == RAIL_TX_POWER_MODE_2P4_HP) {
        if (power >= 200) {
            return RAIL_TX_POWER_LEVEL_HP_MAX;
        } else if (power >= 130) {
            return 93;
        } else if (power >= 30) {
            return 33;
        } else if (power >= 0) {
            return 22;
        } else if (power >= -100) {
            return 7;
        } else {
            return 1;
        }
    }
    if (mode == RAIL_TX_POWER_MODE_SUBGIG) {
        if (power >= 200) {
            return RAIL_TX_POWER_LEVEL_SUBGIG_MAX;
        } else if (power >= 130) {
            return 86;
        } else if (power >= 30) {
            return 26;
        } else if (power >= 0) {
            return 18;
        } else if (power >= -100) {
            return 6;
        } else {
            return 1;
        }
    }
    // return an error value if no PA is initialized
    return RAIL_TX_POWER_MIN;
}

RAIL_TxPower_t RAIL_ConvertRawToDbm(RAIL_Handle_t railHandle,
    RAIL_TxPowerMode_t mode,
    RAIL_TxPowerLevel_t powerLevel)
{
    if (mode == RAIL_TX_POWER_MODE_2P4_LP) {
        if (powerLevel >= RAIL_TX_POWER_LEVEL_LP_MAX) {
            return 0;
        } else if (powerLevel >= 2) {
            return -100;
        } else {
            return 1;
        }
    }
    if (mode == RAIL_TX_POWER_MODE_2P4_HP) {
        if (powerLevel >= RAIL_TX_POWER_LEVEL_HP_MAX) {

```

```
}elseif(powerLevel >= 93){return 130;}elseif(powerLevel >= 33){return 30;}elseif(powerLevel >= 22){return 0;}elseif(powerLevel >= -7){return  
-100;}else{return -250;}}if(mode == RAIL_TX_POWER_MODE_SUBGIG){if(powerLevel >= RAIL_TX_POWER_LEVEL_SUBGIG_MAX)  
{return 200;}elseif(powerLevel >= 86){return 130;}elseif(powerLevel >= 26){return 30;}elseif(powerLevel >= 18){return 0;}elseif(powerLevel >= 6)  
{return -100;}else{return 1;}}  
// return an error value if no PA is initialized  
return RAIL_TX_POWER_MIN;}
```

Direct Mode

Rx Direct Mode on EFR32

This document outlines the Rx Direct Mode feature of the EFR32 Wireless Gecko family. Rx Direct Mode is a feature that allows the user to access the raw symbols received by the radio without packet handling. Direct Mode can be useful when the received data does not conform to the standard packet structure or when the user wants to implement custom packet handling. Although, Tx Direct Mode where the modulator signal is fed into the radio directly for immediate transmission also exists, the current document is restricted to discuss Rx Direct Mode only.

Definition of Direct Mode

In the classical sense, in Direct Mode the radio operates via the GPIO interface, where it outputs (expects) the demodulated (modulating) binary signal stream for the received (transmitted) signal. In a broader sense, however, Rx Direct Mode can be considered as any operational mode where the radio operates with a **continuous, digital, semi-processed** or **raw** signal stream requiring the detection and symbol decoding (in some cases even the demodulation) to be performed in software with limited HW assistance.

Practical Importance - Typical Use Cases

In Packet Mode, losing some data symbols during packet detection and clock synchronization is unavoidable. If the synchronization pattern is too short or not included in the transmitted data, reliable detection becomes impossible. In such scenarios, switching to Direct Mode should be considered.

Access to the radio's internal signals provides the advantage of observing the raw, the unprocessed signal stream -- the same signals the radio uses in conventional Packet Mode. This makes Direct Mode an excellent debugging interface, too.

Also, unconventional demodulation (such as multi-level FSK, AFSK, multi-level PSK, Analog, etc.) or coding (32 bit+ DSSS, Pulse-Width OOK, etc.) schemes can only be implemented in Direct Mode on the EFR32.

Finally, Direct Mode can improve the performance compared to what Packet Mode would be capable of by accessing more low-level information, at higher resolution and rate on the signal not just the demodulated stream. This advantage, however, may cost a tremendous amount of CPU resources.

Direct Mode Classification

Types of Direct Mode

The following list outlines the supported types of Direct Mode, arranged by data integrity and complexity. Higher numbers indicate that the signal is accessed earlier in the receiver chain, resulting in less hardware processing of the signal:

(0. Packet Mode: This is the standard operational mode and not considered a type of Direct Mode;)

1. **Synchronous Direct Mode:** bit-clock synchronized demodulated data provided in a binary interface; depends on how the bit clock is recovered;
2. **Asynchronous Direct Mode:** Oversampled demodulated data provided in a binary interface; heavily depends on the symbol detector configuration;
3. **Demod Data:** Amplitude or frequency of the received signal depending on the modulation scheme used;
4. **IQ Data:** Quadrature samples directly tapped after the RF mixer; independent of the demodulator.

Except for IQ Data, each successive mode simplifies the signal by reducing the amount of information it carries while enhancing conciseness. This simplification makes the signal easier to process and allows for higher achievable data rates as it approaches Packet Mode. IQ Data and Demod Data modes are the most versatile, offering detailed insights into the

signal's physical properties. In contrast, Synchronous and Asynchronous Direct Modes operate on signals that have already passed through the demodulator's slicer, which converts the physical signal properties into binary symbols (0s and 1s).

The Synchronous Direct Mode is captured at the `Actual Rx Baudrate [baud]` . From mode 2 to 4, the radio operates at the Oversampling Rate (OSR), and the sample rate is defined by `Raw Data Sampling Rate [sps]` . These parameters can be found in the Radio Configurator generated log files (`radioconf_generation_log.json`).

We recommend using Packet Mode whenever possible, as this is the intended design of our products. If Packet Mode cannot be used, choose the least complex Direct Mode that meets your application's requirements.

As a general rule of thumb, the following guidelines can help you choose the most suitable type of Direct Mode for your application:

- Synchronous Direct Mode:
 - When robust signal detection cannot be set up in the radio configuration, but the frame can be seen on the synchronized bit stream.
 - For Bit Error Rate (BER) tests.
- Asynchronous Direct Mode:
 - In channel scanning/protocol learning applications where the frequency or the baud rate is not precisely known.
 - In applications where enormous baud rate tolerance is desired.
- IQ or Demod Data:
 - For signal analysis, debugging, or custom demodulation.
 - For applications where the signal is not modulated with a supported modulation scheme.

Interfaces and Availability of Rx Direct Mode Types

Direct Mode type availability and the interface through which the data is provided for further processing depend on the radio configuration and the device generation. The following interfaces are available for Rx Direct Mode types:

1. `Direct-to-Buffer` : may be available for all Direct Modes types;
2. `Direct-to-PRS` : may only be available for binary Direct Modes types.
3. `Direct-to-GPIO` : may only be available for binary Direct Modes types;

Several Direct Mode types are only available if the radio configuration is generated with the Rx Direct Mode option enabled (i.e., not the `DISABLED` option is selected). This option is available on a limited subset of Series 2 device generations (e.g., EFR32xG23). These configurations will be referred to as the `Direct Mode Radio Configuration` hereafter.

Note: The `SYNC` mode is preferred over the `ASYNC` mode, as it provides both `Direct-to-PRS` signals simultaneously.

In `Direct-to-Buffer` mode, RAIL stores the samples into the Rx FIFO. In this mode, all Direct Mode types are available and can be configured through the `RAIL_ConfigData(..., data_config)` API.

The type can be selected via the `data_config.rxSource` field and `data_config.rxMethod` must be set to `FIFO_MODE` .

The following table summarizes the available APIs for each Rx Direct Mode types:

Direct Mode Type	RAIL API	Availability
Synchronous Direct Mode	<code>RX_DIRECT_SYNCHRONOUS_MODE_DATA</code>	Limited to <code>Direct Mode Radio Configuration</code>
Asynchronous Direct Mode	<code>RX_DIRECT_MODE_DATA</code>	Limited to <code>Direct Mode Radio Configuration</code>

Direct Mode Type	RAIL API	Availability	
Demod Data	RX_DEMOD_DATA)	No restrictions	
IQ Data	RX_IQDATA_FILTLSB/RX_IQDATA_FILTMSB	No restrictions	

In this mode, the data flow typically looks like this:

1. The samples captured by the radio are transferred to the Rx FIFO allocated for RAIL.
2. The application reads the samples from the RAIL's Rx FIFO into an application buffer.
3. The application processes the samples or transfers them to a host processor through a communication interface.

The highest available baud rate for `Direct-to-Buffer` mode limited by the interface through which the data is provided for further processing (or potentially even lower if the data is processed within the EFR32's core). While the radio can operate at symbol rates of up to several Msps, it's important to note that each IQ sample consists of 4 bytes, whereas in `Demod Data`, the data is represented with just 1 byte. Additionally, the Oversampling rate (baud rate/symbol rate) can be as high as 10-20.

For example, for a 100 kbps PHY with an OSR of 10, the IQ Data rate would be $4 * 100 \text{ kbps} * 10 * 8 = 32 \text{ Mbps (4 MBps)}$, while the Demod Data rate would be $100 \text{ kbps} * 10 * 8 \Rightarrow 8 \text{ Mbps (1 MBps)}$.

Series 2 `Direct-to-PRS` is only available when a `Direct Mode Radio Configuration` is the active radio configuration.

Direct Mode Type	PRS Signal Name
Synchronous Direct Mode Data	<code>PRS_MODEM_DOUT</code>
Synchronous Direct Mode Clock	<code>PRS_MODEM_DCLK</code>
Asynchronous Direct Mode Data	<code>PRS_MODEM_LRDSALIVE</code>

The baud rate of the `Direct-to-PRS` signals is essentially limited by the peripheral to which the PRS signals are connected.

For more information on how to route PRS signals to GPIOs, see the [Debugging training article](#).

For Series 1 devices, Direct Mode types with binary output (Sync and Async) can be traced onto GPIOs (via DBUS interface). These are available via the `RAIL_ConfigDirectMode()` and `RAIL_EnableDirectMode()` APIs.

Note: `Direct-to-GPIO` mode is not recommended on Series 2 devices as it may not work as expected with certain radio PHYs. Instead, use the `Direct-to-PRS` mode.

Examples

In this section, we provide examples on how to set up the different Direct Mode types and interfaces, either through firmware code implementation or, for a quick testing, using RAILtest commands.

Setting up Direct-to-Buffer Mode

C Code

```
/*
Available data sources:
RX_DIRECT_SYNCHRONOUS_MODE_DATA
RX_DIRECT_MODE_DATA
RX_DEMOD_DATA
RX_IQDATA_FILTLSB RX_IQDATA_FILTMSB
```

```
// Defines the RAIL Rx FIFO length and the application's buffer length
#define RX_FIFO_LENGTH 4096
#define BUFFER_LENGTH 2000 // Select the default data source
#define DEFAULT_DATA_SOURCE RX_DIRECT_MODE_DATA

// Allocate the Rx FIFO __ALIGNED(RAIL_FIFO_ALIGNMENT)
uint8_t rx_fifo[RX_FIFO_LENGTH]; // Allocate a ping-pong style application buffer
uint8_t rx_buffers[2][BUFFER_LENGTH]; // Initialize the active buffer index
uint8_t active_buffer = 0; // .../* Initialization code */{ // ...
    RAIL_Status_t status; // Select Direct Mode type and configure the FIFO_MODE
    RAIL_DataConfig_t data_config = {rxSource = DEFAULT_DATA_SOURCE, rxMethod = FIFO_MODE}; // Configure `Direct-to-Buffer` mode
    status = RAIL_ConfigData(rail_handle, &data_config); if(status != RAIL_STATUS_NO_ERROR){ // handle error} // Set the Rx FIFO threshold for the Rx
    FIFO almost full event RAIL_SetRxFifoThreshold(rail_handle, BUFFER_LENGTH); // Enable the Rx FIFO almost full event
    RAIL_Events_t events = RAIL_EVENT_RX_FIFO_ALMOST_FULL;
    status = RAIL_ConfigEvents(rail_handle, events, events); if(status != RAIL_STATUS_NO_ERROR){ // handle error} // ...} // Bind the Rx FIFO to the RAIL
    handle
    RAIL_Status_t RAILCb_SetupRxFifo(RAIL_Handle_t railHandle){
        uint16_t rx_fifo_size = RX_FIFO_LENGTH;
        RAIL_Status_t status = RAIL_SetRxFifo(railHandle, &rx_fifo[0], &rx_fifo_size); if(rx_fifo_size != RX_FIFO_LENGTH){ // We set up an incorrect FIFO
        size return RAIL_STATUS_INVALID_PARAMETER; } if(status == RAIL_STATUS_INVALID_STATE){ // Allow failures due to multiprotocol return
        RAIL_STATUS_NO_ERROR; } return status; } // * RAIL event handler */{ // ... if(events & RAIL_EVENT_RX_FIFO_ALMOST_FULL) // This event fires whenever
        the Rx FIFO contains more data than configured // in the threshold; BUFFER_LENGTH in this example{
            uint16_t bytes_read; // Read the Rx FIFO into the active buffer
            bytes_read = RAIL_ReadRxFifo(rail_handle, rx_buffer[active_buffer], BUFFER_LENGTH); if(bytes_read != BUFFER_LENGTH){ // handle error}
            Switch the active buffer for the next iteration
            active_buffer ^= 1; // ...}
    }
```

RAILtest Commands

```
rx 0
setRxFifo <fifo_size>
dataConfig fifo fifo <rx_data_source> 0
// rx_data_source can be 1-5; e.g., RX_DIRECT_MODE_DATA is 4
fifoModeTestOptions 1 1
seteventconfig 0x400 0x000
fiforeset 1 1
rx 1
rxFifoManualRead 0 <fifo_size> 0
```

Setting up Direct-to-PRS Mode

C Code

```
#define SYNC_DOUT_GPIO_PORT gpioPortC
#define DCLK_GPIO_PORT gpioPortC
#define ASYNC_DOUT_GPIO_PORT gpioPortC

#define SYNC_DOUT_GPIO_PIN 0
#define DCLK_GPIO_PIN 1
#define ASYNC_DOUT_GPIO_PIN 2

/*
  Channel 0-5 available on GPIO port A/B
  Channel 6-11 available on GPIO port C/D
*/
#define SYNC_DOUT_PRS_CHANNEL 6
#define DCLK_PRS_CHANNEL 7
#define ASYNC_DOUT_PRS_CHANNEL 8

// (...)

// Initialize PRS signals
PRS_ConnectSignal(SYNC_DOUT_PRS_CHANNEL, prsTypeAsync, PRS_MODEML_DOUT);
PRS_ConnectSignal(DCLK_PRS_CHANNEL, prsTypeAsync, PRS_MODEML_DCLK);
PRS_ConnectSignal(ASYNC_DOUT_PRS_CHANNEL, prsTypeAsync, PRS_MODEM_LRDSALIVE);

// Initialize GPIO pins
GPIO_PinModeSet(SYNC_DOUT_GPIO_PORT, SYNC_DOUT_GPIO_PIN, gpioModePushPull, 0);
GPIO_PinModeSet(DCLK_GPIO_PORT, DCLK_GPIO_PIN, gpioModePushPull, 0);
GPIO_PinModeSet(ASYNC_DOUT_GPIO_PORT, ASYNC_DOUT_GPIO_PIN, gpioModePushPull, 0);

// Assigning PRS signals to GPIO pins
PRS_PinOutput(SYNC_DOUT_PRS_CHANNEL, prsTypeAsync, SYNC_DOUT_GPIO_PORT, SYNC_DOUT_GPIO_PIN);
PRS_PinOutput(DCLK_PRS_CHANNEL, prsTypeAsync, DCLK_GPIO_PORT, DCLK_GPIO_PIN);
PRS_PinOutput(ASYNC_DOUT_PRS_CHANNEL, prsTypeAsync, ASYNC_DOUT_GPIO_PORT, ASYNC_DOUT_GPIO_PIN);
```

RAILtest Commands

```
// The signal-source associations are valid for EFR32xG23/25/28 parts
setdebugsignal <gpio_pin_selection: PC0> CUSTOM_PRS 0x29 6 // SYNC DOUT
setdebugsignal <gpio_pin_selection: PC1> CUSTOM_PRS 0x29 5 // DCLK
setdebugsignal <gpio_pin_selection: PC2> CUSTOM_PRS 0x2A 3 // ASYNC DOUT
```

Setting up Direct-to-GPIO Mode

C Code

```
RAIL_Status_t status;

RAIL_DirectModeConfig_t direct_mode_config = {
    // configure
};

status = RAIL_ConfigDirectMode(rail_handle, &direct_mode_config);
if (status != RAIL_STATUS_NO_ERROR)
{
    // handle error
}

status = RAIL_EnableDirectModeAlt(rail_handle, <enable_tx>, true);
if (status != RAIL_STATUS_NO_ERROR)
{
    // handle error
}
```

RAILtest Commands


```
rx 0  
configDirectMode <parameters>  
directmode 1  
rx 1
```

For additional examples, explore our [EFR32 RAIL Application Examples](#) repository on GitHub.

Overview

RAIL PHY Development

The content in this section is related to the working directly with the physical layer using tools and features provided with RAIL in the Proprietary SDK. This is configurable only when using RAIL or Connect. The content includes theory of and basic information on configuring the EFR32 radio, measurement results and recommendations to comply with various standards, as well as guides on how start from scratch or how to set up the radio to be compatible with an existing setup.

- [EFR32 Radio Configurator Guide for Simplicity Studio 5 \(AN1253\)](#): Describes the radio configurator GUI for RAIL framework applications in Simplicity Studio 5. With it, you can create standard or custom radio configurations on which to run your RAIL-based applications. The role of each GUI item is explained.
- [EFR32 Series 1 Long Range Configuration Reference \(UG460\)](#): Introduces the long-range radio profile, describes its development, and examines underlying details that enable it to realize extended range. Instructions for using example applications are included.
- [EFR32 Series 2 Long Range Configuration Reference \(UG460.2\)](#): Describes how to test long range performance on EFR32 Series 2 devices using Simplicity Studio 5 and Silicon Labs development hardware.
- [EFR32 RF Evaluation Guide](#): Describes using RAILTest to evaluate radio functionality, as well as peripherals, deep sleep states, etc. With it you can fully evaluate the receiving and transmitting performance and test RF functionality of development kit hardware or custom hardware.

EFR32 RF Evaluation Guide

AN972: EFR32 RF Evaluation Guide

This section replaces *AN972: EFR32 RF Evaluation Guide*. Further updates to this application note will be provided here.

This evaluation guide provides an easy way to evaluate the performance of the Wireless Gecko EFR32 devices using the Silicon Labs Radio Abstraction Interface Layer (RAIL). RAILtest is a standalone test application, which is used for testing radio functionality, as well as peripherals, deep sleep states, and so on. Basic transmit and receive commands allow customers to fully evaluate the receiving and transmitting performance and to test RF functionality of development kit hardware or customer hardware. Note that some of the lab tests require RF test equipment, such as a spectrum analyzer and RF signal generator. It is a useful test application to be used for basic testing of your custom EFR32 design and characterization. The test can be used in the laboratory to measure the basic RF parameters of the radio (output power, sensitivity, and so on).

This evaluation guide describes how to install the RAILtest application to EFR32 devices and demonstrates how to make frequently-required, conducted transmission and reception tests using the RAILtest application.

Note: Measuring over-the-air range performance of EFR32 devices, please refer to the Range Test application, available in Simplicity Studio. Refer to *UG147: Range Test Demo User's Guide* for its usage.

The following measurements will be covered on the transmission side:

- Tx power
- Phase Noise
- Power Spectral Density Mask
- Spurious Emission
- Transient power

The following measurements will be covered on the reception side:

- Rx sensitivity
- Selectivity
- Blocking
- Intermodulation
- Maximum input power

Proprietary is supported on all EFR32FG devices. For others, check the device's data sheet under Ordering Information > Protocol Stack to see if Proprietary is supported. In Proprietary SDK version 2.7.n, Connect is not supported on EFR32xG22.

Running Simplicity Studio

Running Simplicity Studio

Simplicity Studio 5 and Flex SDK v 3.x introduced a number of new features and enhancements. This chapter describes how to use Simplicity Studio 5 and Flex SDK v3.x as well as the older Simplicity Studio 4 and Flex SDK 2.x for testing.

Before beginning, you should have installed Simplicity Studio and the Flex SDK according to the instructions in the relevant quick-start guide: *QSG138: Proprietary Flex SDK v2.x Quick Start Guide* or *QSG168: Proprietary Flex SDK v3.x Quick Start Guide*.

Before running RAILtest in either version, the test target must be configured according to the following instructions:

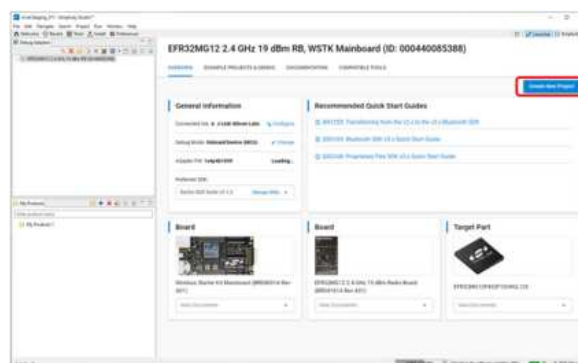
To start, set up an EFR32 development kit radio board with a mainboard for Wireless Starter Kits (WSTK). Once you have installed all the required software you can connect your EFR32 development kit hardware to your PC using a mini-USB cable. Make sure the 3-way power switch in the bottom left is set to AEM.

If you want to connect to your WSTK over Ethernet, plug in an Ethernet cable as well at this time. The IP address will be printed to the LCD screen during startup of the WSTK but may be lost when the app starts. To see this again, reboot the WSTK and press the reset button for several seconds to prevent the EFR32 from loading its application.

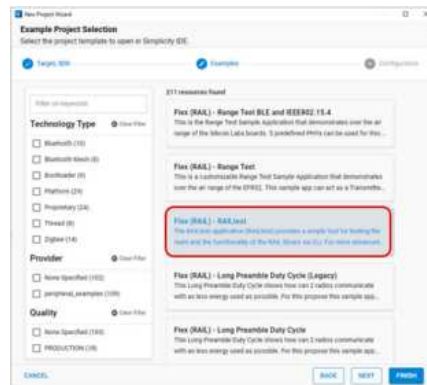
Select RAILtest Application

Start Project In Simplicity Studio 5

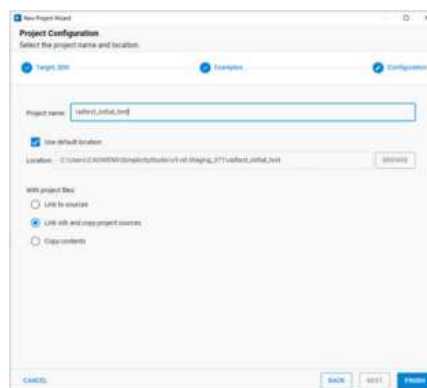
1. When Simplicity Studio 5 opens, select the target in the Debug Adapters view. If you have more than one device connected, select based on the serial number. After connecting the WSTK to the PC, the Start Screen on the LCD shows the device's serial number.
2. Click the **OVERVIEW** tab if it is not already selected.
3. Click **Create New Project**.



4. Select the **Flex (RAIL) - RAILtest** application, and click **Next**. You can use keywords or filters to find the application.

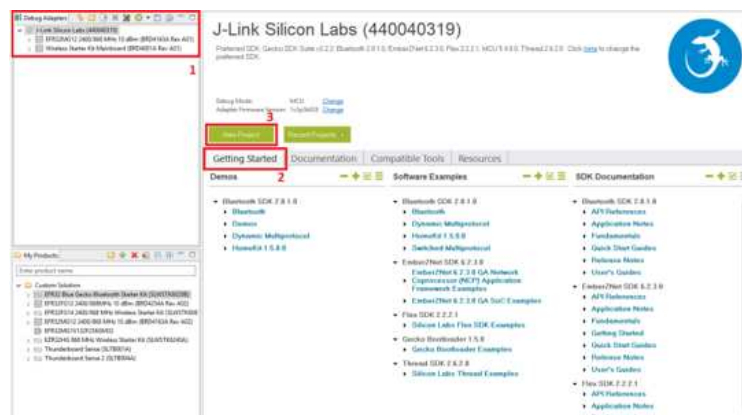


5. Rename the project and/or change the location, if desired. Click **Finish**.

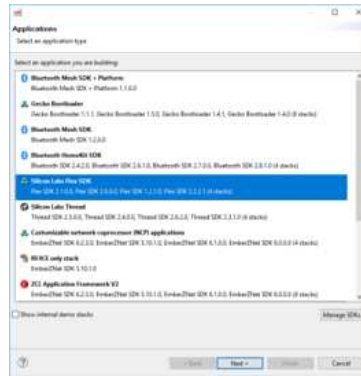


Start Project In Simplicity Studio 4

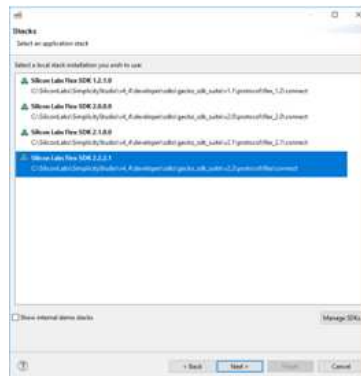
1. When Simplicity Studio 4 opens, select the target in the Debug Adapters view. If you have more than one device connected, select based on the serial number. After connecting the WSTK to the PC, the Start Screen on the LCD shows the device's serial number.
2. In the Launcher (default) perspective, if it is not already selected, click the **Getting Started** tab.
3. Click **New Project**.



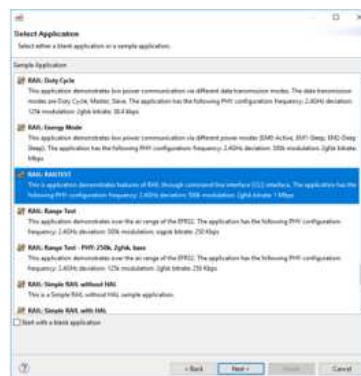
4. Select **Silicon Labs Flex SDK** as the application type.



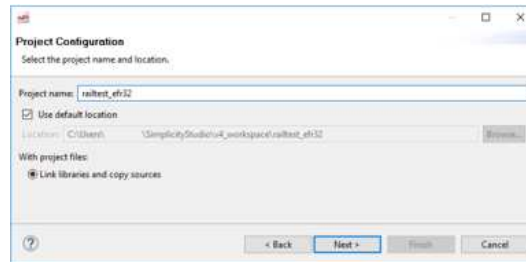
5. Click **Next**.
6. Select the Flex SDK version you would like to use (preferably the latest one).



- Click **Next**.
- Select the **RAIL: RAILTEST** application.



9. Click **Next**.
10. In the Project Configuration dialog, specify a location for your application.

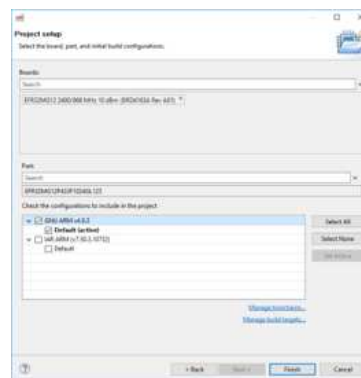


Note: the location must be on the same Windows partition as the stack.

11. Specify a name for your application.

12. Click **Next**.

13. The Project Setup dialog shows the various build configurations available. Check one of the initial build configuration to include in the project. (You can edit these later through the 'Manage Configurations' command.)



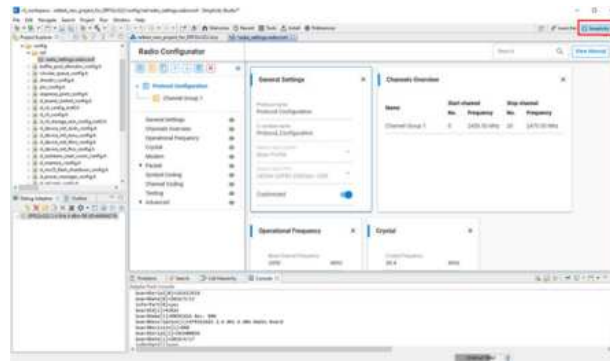
14. Click **Finish**.

Note: You must have a Toolchain and Build target selected and configured for the **Finish** control to enable. If you do not see **Finish** enabled, check your Toolchains and Build targets by clicking the links at the bottom of the dialog.

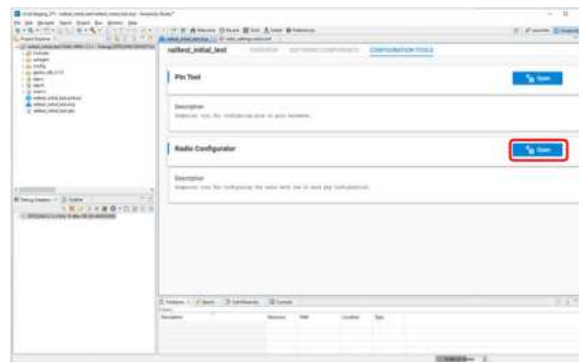
Configure and Generate the Application

Configure and Generate In Simplicity Studio 5

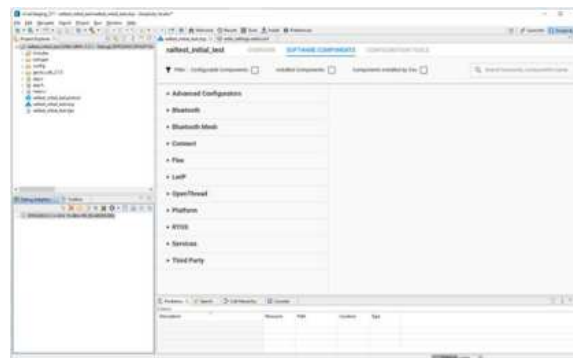
After creating the project, the Simplicity IDE perspective opens. You can switch between Simplicity IDE and Launcher perspectives using the controls in the top right corner. The project explorer view is on the top left. It contains all the files for the opened projects. By default the project's `<project_name>.slcp` and `radio_settings.radioconf` files are already opened. The project is defined in the `.slcp` file. These two files replace Simplicity Studio 4's `.isc` extension file.



If the Radio Configurator is not open, on the project .slcp tab click the **CONFIGURATION TOOLS** tab, and click **Open** on the Radio Configurator card. See *AN1253: EFR32 Radio Configurator Guide for Simplicity Studio 5* for more information.



Projects are configured by installing and uninstalling components, and configuring installed components. The **SOFTWARE COMPONENTS** tab displays categories of components on the left, and details about the selected component on the right. Several filters as well as a keyword search are available to help you explore the various component categories.



Simplicity Studio 5 disables buttons, LEDs and the LCD display in the RAILtest application by default. These features help the user to interact with RAILtest application and are enabled by default in older versions of Simplicity Studio. To enable these features, you may install the following software components: Button 0 (as btn0), Button 1 (as btn1), LED 0 (as led0), LED 1 (as led1) and RAILtest Graphics.

Note: Feature availability may differ on different boards.

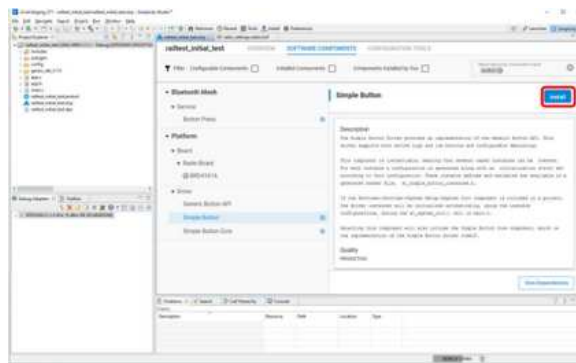
These components are in the following component categories:

- Platform > Driver > Simple Button > btn0 and btn1
- Platform > Driver > Simple LED > led0 and led1
- Platform > Radio > RAILtest, Graphics

Note: These may install more than one dependency, depending on their function.

For example, to install a Simple Button:

1. On the **SOFTWARE COMPONENTS** tab, enter *button* in the search field in the top right corner.
2. Select the Simple Button component and click **Install**.



3. Name the component instance, then click **Done**.



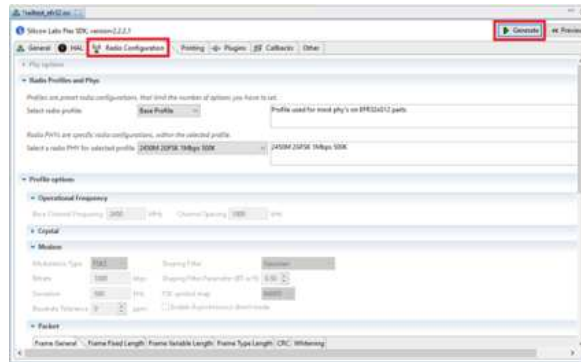
Project files autogenerate as you add, remove, or configure components.

Configure and Generate in Simplicity Studio 4

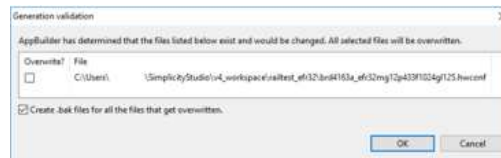
When you finish creating your sample application, an Application Builder General tab opens.



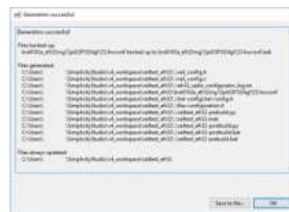
1. In the General tab, if the architecture parameters shown for MCU and Radio and Board Type are not correct for your target device, click **Edit Architecture** to change the parameters. In general, the initial configuration settings for sample applications should be correct.
2. The RAIL application framework allows you to modify the PHY configuration for the application. Select the **Radio Configuration** tab to modify the PHY configuration for your application. For more information on how to set the modem parameters, please refer to *AN971: EFR32 Radio Configurator User's Guide*.



3. Click **Generate** in the upper right corner of the Application Builder window. Note that whenever the Build Configuration has been changed, you have to click **Generate** again.
4. Select if you want to overwrite any of the existing files. Click **OK**.



5. The next dialog shows the generated files. Click **OK** to continue.



Build the Application

In Simplicity Studio 5, either right-click the .slcp file in the Project Explorer view and click **Build**, or with the project open, click **Build** (hammer icon) in the toolbar.

In Simplicity Studio 4, double-click on app_main.c in the Project Explorer view to open it in the editor. Click **Build** (hammer icon) in the toolbar.

Wait until the build process is finished.



Verify that the project has built without any error in the Console view at the bottom of the Simplicity IDE.

Load the Binary onto your Device/Flash Programming

If a full erase is not necessary, click Debug (bug icon) in the menu bar. This flashes the project into the board if it was successfully built. This will only update the program memory.

Once the project is flashed, the board can be started by clicking Resume in the menu bar.

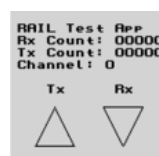
Alternatively:

1. In the Debug Adapters perspective, select the target device.
2. With the device selected, open the Flash Programmer through the Launcher perspective's Compatible Tools tab or by clicking Flash Programmer (chip icon) in the Simplicity IDE menu bar.
3. The Flash Programmer opens. Select the file type.



4. Navigate to the .bin or .hex image you wish to upload.
5. Click **Erase** to make sure that any previous bootloader or other non-volatile data is erased before your new image is uploaded.
6. Click **Program** to program the flash.
7. You will be notified once the upload is complete.

After starting the demo, the screen shows three parameters: Rx Count, Tx Count, and Channel Number.



Serial Port Communication

The RAILtest application should be accessed using a standard serial port terminal. The default version uses the WSTK's built-in serial port to provide access to this, so do not attempt to connect directly to the UART GPIOs.

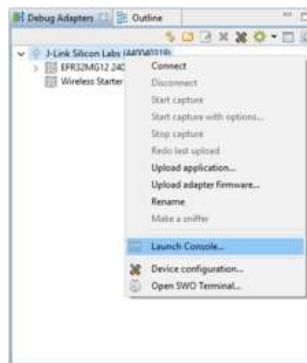
The serial port can be accessed using Ethernet or USB. Both provide the same level of functionality so it is completely up to you what to use.

- For USB, plug the device into your computer and find the Virtual COM port that is created. It will show up as a "JLink CDC UART Port" in the Device Manager. The Virtual COM port should work with any UART settings, but the physical UART is configured by default for 115200 8-N-1 if you want to match that. For UART-based communication, you can use any terminal emulator program that supports serial port communication, like TeraTerm or PuTTY.
- Simplicity Studio contains a built-in terminal as well. To run it, right-click on the device on the Debug Adapters view and click **Launch Console**. The RAILtest API should be accessible from the Serial 1 port.
- For Ethernet, use a telnet client to connect to port 4901 on the WSTK. You can use a program like PuTTY for this or the standard command line telnet client.

Console View

Once the RAILtest application is loaded onto the EFR32 you can interact with the RAILtest application using the Command Line Interface. The Console View will give you a CLI interface in Simplicity Studio's Network Analyzer Perspective so that you can interact directly with the RAILtest application.

1. Right-click the device in the Debug Adapters view.
2. Select **Connect** (if you are not already connected) and then **Launch Console**. To get a prompt on the Console Serial 1 tab, press **Enter**.



Basics of the RAILtest Application

Basics of the RAILtest Application

The RAILtest application provides a simple tool for testing the radio and the functionality of the RAIL library. For any advanced usage you must write your own software against the RAIL library and create a custom radio configuration. The RAILtest application is further described in *UG409: RAILtest User's Guide*.

Note: If you are going to use the WSTK as a mobile device to run the RAILtest application, it is recommended that you connect an external AA battery pack or a USB power bank to the WSTK board. The coin cell battery will not have enough power to do long-term testing using sample applications like RAILtest.

CLI

The most powerful way to interact with the sample application is using the provided command line interface. The setup instructions in earlier sections describes how to connect to this interface.

Command Input

The syntax for this interface is the standard command [arg0, arg1, ...] syntax, where the number and type of arguments depend on the specific command. Numeric values can be prefixed with 0x to indicate hexadecimal values. Note that the maximum number of arguments to any command is 9 and the maximum length of a command line is 255 characters.

Use the `help` command to see a full listing of available command options.

Command Output

All responses to commands are formatted in a human readable yet parsable format. There are two variations of this format: single- and multi-line. Both of these follow these rules.

- Start and end with curly braces { }.
- List the command name, enclosed in parentheses ().
- Contain any number of tag/value pairs enclosed in curly braces { }.
- Carriage returns and line feeds are treated as whitespace by any parser.

Single Response

Used when there is only a single response to a command.

- There is a single start/end curly brace wrapper.
- Tag/value pairs are wrapped in a single set of curly braces, separated by a colon {tag:value}.

Example:

```
> getchannel
{{{getchannel}}{channel:4}}
```

Multi Response

Used when a command may have multiple responses, for example when reading a block of memory or receiving multiple packets.

- Response starts with a header, delimited by a hash # at the start of the line.
- Header includes the command name, followed by any tags individually wrapped with curly braces { }.
- Following the header, any number of responses can be provided.
- Data lines do not contain the command name or tags, only the values that correspond to the tags in the order described in the header.

Example:

```
> getmemw 0x20000000 4
#{{(getmemw)}{address}{value}}
{{0x20000000}{0x0000e530}}
{{0x20000004}{0x000051c6}}
{{0x20000008}{0x0000c939}}
{{0x2000000c}{0x0000e090}}
```

Transmission Test

Transmission Test

Transmitted Signal Types

The device is capable to generate continuous unmodulated signal (CW mode, also referred as single tone or sinus wave signal), packets (packet mode) and a continuous pseudo random stream (PN9 mode). The following table helps to decide which transmission mode should be used for a corresponding measurement.

	CW mode	PN9 mode	Packet mode
Tx Power	x	—	—
Power Spectral Density Mask	—	x	—
Phase Noise	x	—	—
Spurious Emission	x	—	—
Transient Power	—	—	x

The following sections describe how to turn these modes on, and how to modify some corresponding parameters to customize the output signal.

Output a Continuous Unmodulated Tone

CW mode is used to test the output power of the test card and observe the unmodulated carrier spectrum. In this mode, only the frequency and the output power are configurable parameters. There are two modes for CW generation. One is designed to generate CW signal [tone] with the best possible quality. The other is specifically designed to measure phase noise of the device [phaseNoise]. For more information on phase noise measurement, see [Phase Noise Measurement](#). The difference between these two modes of CW signal generation is the PLL bandwidth settings. In the [phaseNoise] CW mode, the PLL bandwidth is the same as the current radio settings. With this method the generated CW signal provides information on how the PLL behaves when transmitting packets during operation.

Note: Before setting power and some other parameters, the device must be in idle receiver mode, by calling rx 0 .

Set Receiver to Idle:

```
rx 0 {{{(rx)}{Rx:Disabled}{Idle:Enabled}{Time:539309681}}}
```

Enable CW Mode:

```
setTxStream 1 0 [enable] [streamMode] Enable(1 0) or Disable(0 0) a CW tone from the radio.
```

The command above also sets the PLL bandwidth to narrower than it's set on the actual radio configuration to produce better quality CW tone.

Enable CW Signal for Phase Noise Measurement:

```
setTxStream 1 3 [enable] Enable(1 3) or Disable(0 3) a tone from the radio with the same PLL bandwidth as the current PHY configuration.
```

Set Tx Power:

The `setPower` command can be used to change the output power; enter the desired power in deci dBm.

Note: You can check the Tx power limitation with the `getPower` command. The radio state must be IDLE for Rx and Tx side. Call `rx 0` and `SetXxTone 0` first.

```
setPower s [power] Set the current transmit power in deci dBm
```

Set Channel:

A way to change the frequency is to change the channel. The specific channel configuration depends on the PHY configuration you have chosen for your test app. To switch between channels, use the `setChannel [num]` command. In addition, you can use button PB1 to cycle between channels. (The channel spacing value is used for relative frequency configuration to configure a frequency so many channel spacing away from the base channel frequency.)

To get the current radio channel, use this command:

```
getchannel Get the current radio channel
```

To set another radio channel, use this command:

```
setchannel u Set the current radio channel
```

Set Frequency:

To modify the frequency to a value not defined in the channel list, set the application into the FREQUENCY_OVERRIDE debug mode via `debugMode`, which tells the application to ignore the current channel selection. Once in the FREQUENCY_OVERRIDE debug mode, you can use the `freqOverride` command to switch to another center frequency.

Note: The `freqOverride` command requires you to be in FREQUENCY_OVERRIDE debug mode.

Note: The radio state must be IDLE in order for the channel to be changed or the frequency to be modified. Call `rx 0` first.

Example:

```
rx 0 [enable] Enable(1) or Disable(0) receive mode
setDebugMode 1 [mode] 1 = Frequency Override, 0 = Disable debug mode
freqOverride 240400000 [freq] Change to freq specified in Hz. Requires debug mode to be enabled.
```

Set Different PHY Configuration:

To switch between PHYs in a multi PHY configuration, use the `setConfigIndex` command.

Example:

```
> setConfigIndex 0
>{{(setConfigIndex)}}{configIndex:0}{firstAvailableChannel:0}}
```

Output a Continuous Modulated PN9 Stream

In this mode, the modulation content is generated internally using a pseudo-random (PN9 sequence) bit generator. The primary purpose of this mode is to observe the modulated spectrum without having to provide data for the radio. This mode is usually used to test the power spectral density mask.

In this mode, only the frequency and the output power are configurable parameters. To change these parameters, refer to [Output a Continuous Unmodulated Tone](#).

Enable PN9 mode:

```
setTxStream    Enable/Disable a PN9 stream from the radio
```

Output Packets

The application starts in packet mode with the receiver enabled. In this mode we receive and transmit packets using the radio's frame controller hardware.

To transmit a specified number of packets, use this command:

```
tx    w    [n] Transmit n packets. If n is 0 transmit infinitely
```

Or press button PB0. If you hold PB0 for a couple of seconds or run the `tx 0` command, you can toggle the continuous transmit mode.

In this mode, the frequency, the output power, delay in between each transmitted packet, the length, and the value of the bytes to send are configurable parameters. To change the frequency and the output power parameters, refer to [Output a Continuous Unmodulated Tone](#).

When transmitting multiple packets or infinite packets there is a configurable delay in between each transmit. By default this is 250 ms, but it can be set with the following command:

```
setTxDelay    w    [delay] Set the inter-packet delay in milliseconds for repeated Tx
```

Get the inter-packet delay in milliseconds for repeated Tx, use the `getTxDelay` command.

If the desired delay is lower than you can set with `setTxDelay`, all the peripherals must be disabled to improve performance.

```
setPeripheralEnable    u    [enable] Turn LEDs and LCD on/off
```

The application by default sends a fixed packet, but it is possible to override the values via `setTxPayload`. The command allows you to modify the values of the payload at specific offsets. For instance to modify the first 4 bytes sent in the packet to be 0x01 0x02 0x03 0x04, this is the command:

```
setTxPayload 0 0x01 0x02 0x03 0x04
```

Note: To view the currently configured TX Packet information, use the following command:

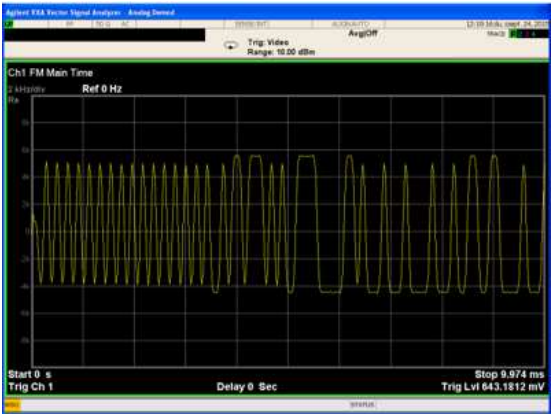
```
printTxPacket    Print the current Tx data and length
```

A byte in the payload holds the length of the packet for variable length schemes. The length byte in IEEE 802.15.4 is the first byte; this is what you can set using the `setTxPayload` command. The length byte also covers the CRC length of the packet, and the length parameter covers the length byte; this is what you can set using the `setTxLength` command.

All in all, here is an example to set 3-byte payload to 0x01 0x02 0x03, while using IEEE 802.15.4 configuration:

```
setTxLength 4
setTxPayload 0 0x05 0x01 0x02 0x03
```

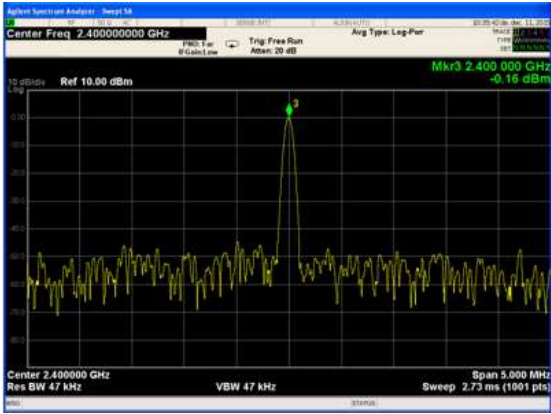
A spectrum analyzer is used to view the over the air sent packets of the radio. It must be connected through a proper RF cable to the RF connector of the radio board.



Tx Power Measurement

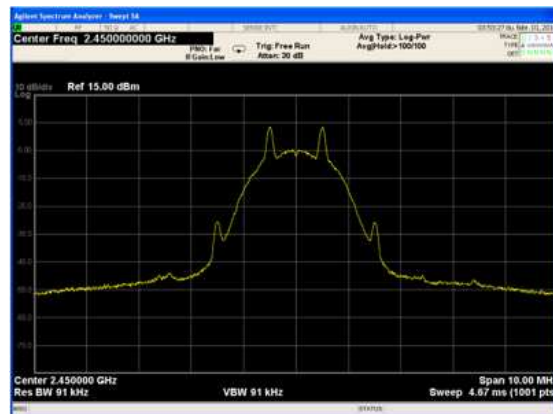
A common way to measure the transmitted power of a device is to transmit a CW signal, then measure the transmitted signal with a spectrum analyzer or an RF power meter device. The measurements should be conducted, and proper RF cabling is necessary.

A spectrum analyzer is used to measure the transmit performances of the radio and must be connected through a proper RF cable to the RF connector of the radio board. In this example the transmitted power is measured in the CW signals 5MHz frequency range and the CW signal is marked with a marker (Mkr3). The transmitted power is displayed on the right upper corner, where the marker is placed.



Power Spectral Density Mask Measurement

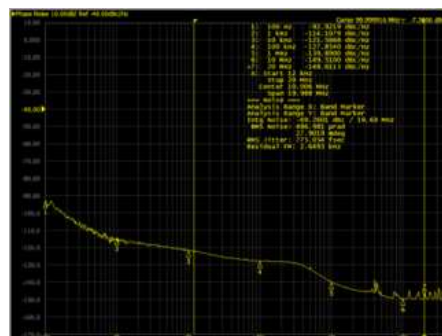
Power Spectral Density (PSD) gives feedback about the transmitted signal's spectrum usage. PSD is usually measured with a PN stream. However, Silicon Labs recommends measuring the device's PSD by enabling the PN9 stream. A spectrum analyzer is used to measure the power spectral density mask of the radio and must be connected through a proper RF cable to the RF connector of the radio board. RF cable loss should be compensated for accurate Tx power output level measurement. The proper reference level and span of the device is required as well.



The figure above shows the demodulated signal when the radio board works in PN9 mode. The data rate is 1 Mbps; the deviation is 500 kHz, and the modulation type is 2GFSK.

Phase Noise Measurement

Phase noise (PN) is a frequency domain representation of a signal's jitter. Phase noise is usually measured when a reference signal's quality is specified. PN measurement gives a feedback on how the - ideally - stable reference signal or other CW signal's frequency changes in a short period of time. PN is usually represented as a frequency-amplitude graph or a notable point - from the graph - is presented in dBc/Hz.



Phase noise is measured with a spectrum analyzer, optimally one with a phase noise measurement feature. Proper connectors and cables are obligatory for this measurement. The PN graph usually contains only one side of the center frequency, as both sides are nearly identical. Frequency is usually represented logarithmically.

EFR32 devices' phase noise is usually measured, giving a feedback of the carrier's quality. If PN is measured on an EFR device, a CW signal is transmitted for measuring. PN not only effects the Tx performance but also the Rx side. The PN gives feedback on the oscillator quality, hence the transmit side and receiver side are affected as well. High PN on the transmitter side results in bad EVM and on the receiver side results in poor sensitivity, as PN is added to the noise level.

Note: Use RAILtest's special CW mode for phase noise measurement described in [Output a Continuous Unmodulated Tone](#).

Spurious Emission Measurement

Spurious emissions are unwanted emissions at frequencies other than those of the transmitted channel. Spurious emission measurement is done by measuring the observed (transmitted) channel and its' neighbor while transmitting packets. A spectrum analyzer is used for measuring and at each frequency at which a spurious component is detected, the power level

is measured and noted. The measured frequency bandwidth is typically 2-10X as the occupied channel bandwidth. Some RF spectrum analyzers have a spurious emission measurements feature, to make performing these measurements easier. Spurious measurements are often taken while standard packets are transmitted, but some region's standard requires a different type of transmitted signal.

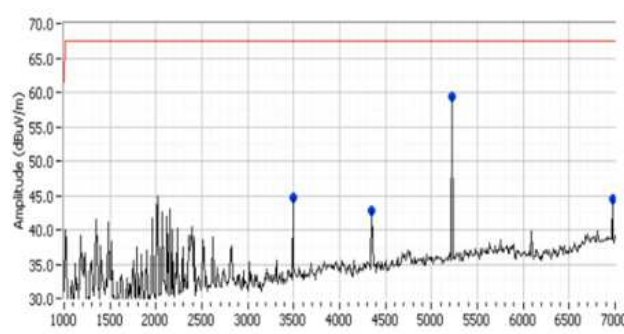


Figure 3.5. Spurious Emission Occurs as Spikes in the Measured Spectrum

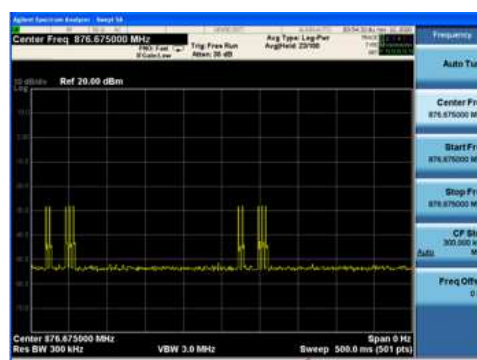
Transient Power

Transient power is power appearing in frequencies other than the occupied channel when the transmitter is turned on or off. Transient power is expressed in dBm and measured at specific offset frequencies from the carrier. Standards usually describe offset frequencies with a maximum tolerated power of transient power.

Transient power can be measured with a spectrum analyzer, used in zero span mode. Zero span mode is used to measure a specific frequency. By changing the resolution bandwidth parameter of the analyzer, the measured frequency region can be set. A higher resolution bandwidth enables acquiring a wider frequency region, and allowing more power to the analyzer.

The EFR32 device is used in packet mode and sent at least 5 to 10 packets. The spectrum analyzer must be set to measure all of them with one measurement, by changing measurement / sweep time. After measurement the peak value must be recorded.

The following is an example of the measurement.



The first figure is a measurement at 78 kHz and the second is at 1275 kHz offset from the carrier frequency of the signal. The signal's power is 16 dBm. The measurement shows that at both settings the maximum power is around -28 to -30 dBm. It is important to take in account that the resolution bandwidth (RBW) is not the same at the two measurements and this should be compensated in some cases. In this case, the power density at the second measurement must be less than the first one, but a wider frequency range is measured. Larger RBW result in more power being received at the wider frequency range.

Enable Direct Mode

In direct mode, the radio will still attempt to decode received packets, but the data bit streams are input to and output from the chip in real-time on a physical I/O pin. This is often useful in legacy systems that do not use a conventional packet structure, or when the host MCU must perform customized packet handling or processing. In TX Direct mode, the TX modulation data is applied to an input pin of the chip and processed in 'real time' (that is, not stored in a register for transmission at a later time).

The GPIOs for direct mode are fixed for now to the following pins.

```
DIN - EFR32_PC10 -> EXP_HEADER15/WSTK_P12
DOUT - EFR32_PC11 -> EXP_HEADER16/WSTK_P13
```

The data on these pins is asynchronous and can be connected directly to a UART. To enter direct mode issue the `directMode 1` command after starting the app. To leave direct mode use `directMode 0`. To transmit, enable the transmitter by issuing `directTx 1` and later stop it with `directTx 0`. Receive is controlled using the standard `rx 1/0` command, but is enabled by default when not transmitting.

In this mode, only the frequency and the output power are configurable parameters. To change these parameters, refer to [Output a Continuous Unmodulated Tone](#).

Reception Test

Reception Test

Packet Error Rate (PER) and Bit Error Rate (BER) are error ratios that could measure various reception tests

	PER	BER
Rx sensitivity	triggered and non-triggered	x
Selectivity	x	x
Blocking	x	x
Intermodulation	x	x
Maximum input Power	x	x

EFR devices have a special mode to measure BER, with additional functions that can be used to calculate and measure BER easily. However, Silicon Labs does not recommend using this mode for RF evaluation, as regular frame detection does not occur in this mode. Instead, measuring PER with the normal operational mode turned on is recommended. Measuring PER gives more accurate feedback about how the EFR radio configuration perform, as it is testing in a mode that will be operational in the field after deployment.

PER Measurement

Basics of the Packet Error Rate

To measure PER, the accurate packet symbols must be known. PER [%] is calculated by the following:

$$PER = \frac{P_{Error}}{P_{Sent}} * 100$$

where

- Error = the number of packets not received correctly
- Psent = the number of packets sent and the result is displayed as a percentage.
- PER measurement requires an accurate device that is able to send the packets the receiver under test (RUT) expects to receive. An RF vector signal generator is a good fit for these measurements.
- Typical PER measurement procedure using RAILtest:
 1. Record the packet that should be transmitted to the RUT (receiver under test) and load it to the RF signal generator.
 2. Connect the RF signal generator and the RUT with the proper RF cable.
 3. Send 100 to 1000 packets to the RUT. (P sent)
 4. View the received number of packets by using `status` (RAILtest) command for example. The `RxCount` value gives the successfully received number of packets. (P sent - packets received = P error)
 5. Calculate PER.
- Using the `status` command, `RxCount` is the number of successfully received packets. `SyncDetect` is the number of sync word detection and `FrameErrors` is the number of CRC errors acquired.

Record a Transmitted Packet

To know how a packet is assembled on an EFR32 device, it is practical to record a transmitted packet. A packet can be captured and demodulated with an RF vector spectrum analyzer from the EFR32 device's RF port, or the transmitted symbols could be wired to a GPIO pin before being sent to the modulator. The second option offers an easy way to see the transmitted baseband packet symbols before the modulation and upconversion to RF frequencies.

To port the transmitted packet symbols to the GPIO port, the PRS channels should be configured by using the `setDebugSignal` command. Use the `setDebugSignal` help command to learn more about the function. For additional information about PRS channels, see the Silicon Labs knowledgebase about *RAIL Tutorial: Debugging* or the EFR32 page of the RAIL API documentation.

For example (using EFRxG13):

```
setDebugSignal PC11 TXACTIVE > PC11 GPIO pin active when TX is active
```

```
setdebugsignal PB2 CUSTOM_PRS 0x2a 0x5 > Data Clock is routed to PB2 GPIO
```

```
setdebugsignal PB3 CUSTOM_PRS 0x2a 0x6 > Data (Tx packet) is routed PB2 GPIO
```

During transmission, the data clock and data is recorded on the respective GPIO pins:



Non-Triggered PER

By default the application starts in packet mode with the receiver enabled. In this mode we receive and transmit packets using the radio's frame controller hardware. If a packet is received, the payload content and length, the RSSI, the CRC status, and the time can be read from the receiver response.

Example:

```
{{(rxPacket)}{len:16}{timeUs:290402914}{timePos:4}{crc:Pass}{rssi:-50}
{qi:240}{phy:0}{isAck:False}{syncWordId:0}{antenna:0}{channelHopIdx:254}
{payload: 0x0f 0x0e 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 0x99 0xaa 0xbb
0xcc 0xdd 0xee }}
```

This test also involves an RF signal generator, which can transmit predefined packets. In this mode, the radio is set to receive and the generator must send packets. If the radio does not receive the packet, there is no RX response. To view the number of received packets, CRC errors, and sync detect errors, use the status command.

Example:

```
{{(status)}{UserTxCount:38}{AckTxCount:0}{UserTxAborted:0}{AckTxAborted:0}{UserTxBlocked:0}
{AckTxBlocked:0}{UserTxUnderflow:0}{AckTxUnderflow:0}{RxCount:0}{SyncDetect:8}{NoRxBuffer:0}{RfSensed:0}
{ackTimeout:0}{ackTxFpSet:0}{ackTxFpFail:0}{ackTxFpAddrFail:0}{RfState:Rx}{RAIL_state_active:0}
{RAIL_state_rx:1}{RAIL_state_tx:0}{Channel:0}{AppMode:None}{TimingLost:0}{TimingDetect:0}{FrameErrors:8}
{RxFifoFull:0}{RxOverflow:0}{AddrFilt:0}{Aborted:0}{RxBeams:0}{DataRequests:0}{Calibrations:1}
{TxChannelBusy:0}{TxClear:0}{TxCca:0}{TxRetry:0}{UserTxStarted:38}}
```

To reset all the counters use:

```
resetCounters    Resets the Tx and Rx counters
```

If a long and fast test needs to be run, it is better to disable the Rx responses, to make the communication and the test faster.

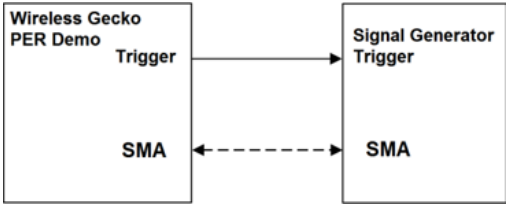

```
setNotifications setRxNotification [enable] Enable(1) or Disable(0) status prints that happen asynchronously (rxPacket, txEnd, txError)
```

Triggered PER

RAILtest can be used to determine the packet error rate (PER) for a given setup. The Wireless Gecko is capable of triggering a connected RF signal generator to transmit a specific number of packets with a configured trigger interval, counting the correctly received packets, and calculating the actual PER value.

To set up this test, a piece of test equipment needs to be configured to send a packet on the rising edge of a GPIO. That should be connected to PC7 on the EFR32, or PC3 on the WSTK. `perRx 100 10000` will configure the Packet Error Rate test to send 100 packets, waiting 10000 μ s between each packet. At the end of the test, the app will give an output indicating that PER mode has finished, and the statistics on the test can be recovered with `perStatus` and `status`. Note that calling `perRx 0 0` will cancel an ongoing test, and that calling `perRx` will have the same effect as calling `resetCounters`.

By adjusting the output power of the generator, the sensitivity of the radio can be determined. It is usually defined for a 1% or 20% packet error rate.



BER Measurement

Running Bit Error Test from the CLI

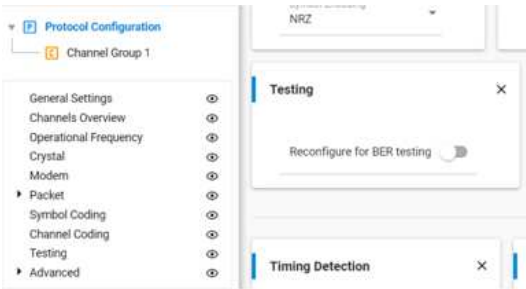
The sensitivity of the radio can be measured with a continuously-looped PN sequence ($x^9 + x^5 + 1$). This mode is called bit error rate (BER). The EFR32 hardware has the ability to enter BER receive mode for diagnostic purposes. The radio is set into continuous receive mode. The RF data needs to be fed to the RX SMA of the test card or radio board. By adjusting the output power of the signal generator or an EFR32, the sensitivity of the radio can be measured (it is typically measured for 10^{-3} BER). See *UG409: RAILtest User's Guide* for further information about BER testing mode.

BER Measurement Procedure

To run the BER test successfully, a radio configuration specific to BER mode must be generated by the radio configurator. To enable BER testing mode by default on reset or startup:

In Simplicity Studio 5:

In the Radio Configurator, on the General Settings card enable **Customized**, and then on the Testing card enable **Reconfigure for BER testing**.



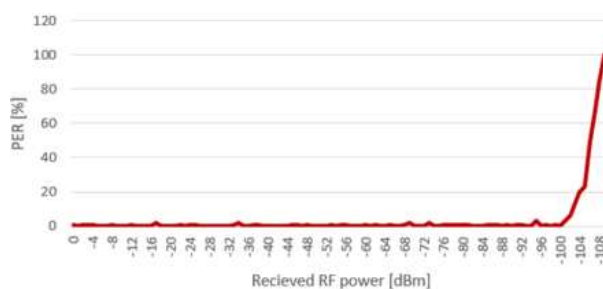
In Simplicity Studio 4:

Check **Reconfigure for BER testing** on the Radio Configuration tab.



Sensitivity Measurement

Sensitivity of a device is the minimum level of received RF power that the device is capable of receiving. Sensitivity is often expressed in dBm. It is usually defined for a 1% or 20% packet error rate, which means that at the sensitivity level (in dBm) the receiver is capable to produce 1% or 20% PER. By adjusting the output power of the generator, the sensitivity of the radio can be determined.



From this graph it can be determined that the sensitivity of a device at 20% PER is -104 dBm. In practice the sensitivity of a device depends on the radio configuration, such as modulation, deviance, and frequency.

To measure the sensitivity of a receiver, PER must be measured at various receiver power levels, in practice between 0 to -120 dBm values with 1 to 5 dB steps.

The measurement results are more accurate if the RF signal generators power steps finer and more packets are sent at each power step. In the above example, the PF power step is 1 dB and 100 packets were sent at each frequency step.

Other Receiver Measurements

The measurement specifications may differ based on the standards for different regions, and are described in more detail in other documents.

Selectivity

Selectivity is a measure of the capability of the receiver to operate correctly while an unwanted signal is present at an adjacent channel.

Measurement procedure: Two generators are used for this measurement. One is generating packets on the operating channel at the sensitivity level of the receiver +3 dB. The second generator is generating an unmodulated signal on the nearest adjacent channel. The maximum power of the second generator at which the RUT (Receiver Under Test) is not able to receive the packets of the first generator is the selectivity, expressed in dBm.

Blocking

Blocking is a measure of the capability of the receiver to operate correctly while an unwanted signal is present at frequencies other than at an adjacent channel or spurious responses are.

Measurement procedure: Two generators are used for this measurement. One generates packets on the operating channel at the sensitivity level of the receiver +3dB. The other generates an unmodulated signal on a specified frequency offset,

based on the local region's specification. The maximum power of the second generator at which the receiver under test is not able to receive the packets of the first generator is the blocking level, expressed in dBm.

Intermodulation

The intermodulation response is a measure of the ability of the receiver to receive a wanted modulated signal, without exceeding a given degradation due to the presence of two or more unwanted signals with a specific frequency relationship to the wanted signal frequency.

Maximum Input Power

Maximum input power is a measure of the maximum power at which the receiver is able to receive the packets without degradation of the signal quality.

RSSI Curve

Received signal strength indicator (RSSI) is an estimate of the signal strength in the channel to which the receiver is tuned. The RSSI value can be read with 0.25 dB resolution per bit. The RSSI may be read at any time while the radio is in Receive mode. The RSSI is not latched, but continuously updated while in Receive mode.

To read the RSSI value, use the following command:

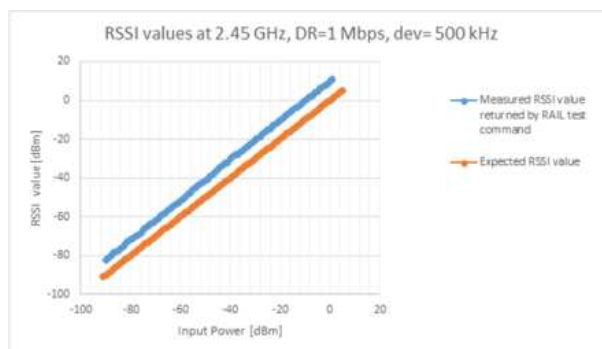
```
getRssi    Get RSSI in dBm if the receiver is turned on.
```

Example:

```
> getRssi
getRssi
>{{(getRssi)}}{rssi:-94}}

> >
```

Note that a systematic offset (see figure below) will appear in the RSSI value returned by the RAILtest command due to matching network, radio configuration, and so on. You must profile the board and account for the offset when using the returned value.



Customer Production Test: Transmission Power

The high resolution RSSI enables accurate channel power comparison measurement that can be useful for production tests transmission power without an instrument.

Measurement procedure:

1. Load the RAILtest application on two EFR32 nodes.
2. Set both nodes to the same channel with the `setchannel n` command, where "n" is the channel number.
3. Issue the `setTxTone` command on the first node to set CW mode.

4. Read the RSSI level on the second node with the `getRssi` command.

Silicon Labs RAIL API Reference Guide

RAIL Library

Silicon Labs Radio Abstraction Interface Layer (RAIL) is a library that can be used as a generic interface for all Silicon Labs radio parts. By programming against this API, you can write code that easily ports across different radio parts while having access to hardware acceleration wherever possible.

Introduction

The RAIL library is mostly standalone with a few external dependencies. Simplicity Studio application builder is not required but is recommended because it simplifies building applications using RAIL. Depending on the hardware platform that you're using, you may be required to provide certain HAL functionality for the library to work properly. These functions are described in hardware-specific documentation sections and valid implementations can be found as a part of the provided emlib/emdrv HAL layers. It is recommended that you use these versions for maximum support, but you're free to re-implement them if necessary.

Using RAIL

Features

At a high level, the functionality supported by RAIL is shown below. For more specifics on the APIs and how to use them, see the module documentation.

- [General](#)
 - Initialize the RAIL API layer.
 - Collect entropy from the radio (if available).
- [Radio Configuration](#)
 - Configure the radio frequency, packet format, channel configuration and other PHY parameters.
 - Query current PHY data rates and parameters like current channel.
- [State Transitions](#)
 - Configure automatic radio state transitions and transition timings.
- [Auto-Ack](#)
 - Configure the radio for automatic acknowledgments.
 - Load the auto ACK payload.
- [System Timing](#)
 - Get the current time in the RAIL timebase.
 - Configure a timer to trigger an event callback after an absolute or relative delay.
 - Specify where within a packet its timestamp is desired.
- [Events](#)
 - Configure which radio or RAIL events the application wants to be told about via callback.
- [Data Management](#)
 - Allows the application to choose the type of data and the method of data interaction through RAIL.
- [Receive](#)
 - Configure receive options like CRC checking.
 - Start or schedule when to receive.
 - Enable and configure [Address Filtering](#) for each packet.
 - Enable and configure [RX Channel Hopping](#) for receiving packets across several channels.
- [Transmit](#)
 - Configure the power amplifier ([Power Amplifier \(PA\)](#)) and set transmit power output.
 - Load and send packet data, either immediately, scheduled, or using CSMA or LBT.
 - Control per-transmit options like CRC generation, ACK waiting, etc.
- [Antenna Control](#)

- Configuring multi-antenna selection.
- [Multiprotocol](#)
 - Manage time-sharing of the radio among different protocols.
- [Calibration](#)
 - APIs for handling various radio calibrations for optimal performance.
- [RF Sense](#)
 - Enable RF energy sensing of specified duration across the 2.4 GHz and/or Sub-GHz bands (EFR32 only).
- [PTI Packet Trace](#)
 - Configure Packet Trace pins and serial protocol.
 - Specify the stack protocol to aid network analyzer packet decoding.
- [Front End Module Voltage Detection](#)
 - Configure RAIL to measure the voltage on a Front End Module during the transmit packet.
- [Assertions](#)
 - Detecting and handling fatal runtime errors detected by RAIL.
- [Diagnostic](#)
 - Output debug signals like an unmodulated tone and a continuously modulated stream of data.
 - Configure crystal tuning for your radio.
 - Fine-tune the radio tuner frequency.
- [Features](#)
 - Macros to indicate which features exist on which chip families.

[Protocol-specific](#) hardware acceleration (each only on supported chips):

- [BLE](#)
 - Configure the Bluetooth Low Energy PHYs available for your device (1 Mbps, 2 Mbps, Coded PHY).
 - Preamble, sync word and whitening adjustment function for connections.
- [IEEE 802.15.4](#)
 - Configure the IEEE802.15.4 2.4 GHz PHY.
 - Configure node address and address filtering for IEEE 802.15.4.
 - Configure auto ACK for IEEE 802.15.4.
- [Sidewalk Radio Configurations](#)
 - Configure 2GFSK 50 kbps Sidewalk PHY.
- [Wireless M-Bus](#)
 - Configure simultaneous M2O RX of T and C modes feature.
- [Z-Wave](#)
 - Configure the Z-Wave PHY appropriate for your region.
 - Configure your NodeId and HomeId for receive filtering and wakeup beam filtering.

Getting Started Example

Below is a simple example to show you how to initialize the RAIL library in your application. Notice that this requires the channel configurations, which can be generated from the radio calculator in Simplicity Studio. By default, RAIL interacts with data on a per-packet basis. For more information, see [Data Management](#).

Note: Before the radio can transmit, ensure the PA is initialized for your platform, which is described in the [Hardware-specific Configuration](#) section below.

```

#include "rail.h"
#include "rail_config.h" // Generated by radio calculator

#define TX_FIFO_SIZE (128) // Any power of 2 from [64, 4096] on the EFR32

static RAIL_Handle_t gRailHandle = NULL;
static RAIL_TxPower_t txPower = 200; // Default to 20 dBm
static uint8_t txFifo[TX_FIFO_SIZE];

static const RAIL_TxPowerConfig_t railTxPowerConfig = { // May be const
    // ... desired PA settings
};

static void radioConfigChangedHandler(RAIL_Handle_t railHandle,
    const RAIL_ChannelConfigEntry_t *entry)
{
    bool isSubGig = (entry->baseFrequency < 1000000000UL);

    // ... handle radio configuration change, e.g., select the desired PA possibly
    // using isSubGig to handle multiple configurations
    RAIL_ConfigTxPower(railHandle, &railTxPowerConfig);

    // The TX power must be reapplied after changing the PA above.
    RAIL_SetTxPowerDbm(railHandle, txPower);
}

static void radioEventHandler(RAIL_Handle_t railHandle,
    RAIL_Events_t events)
{
    // ... handle RAIL events, e.g., receive and transmit completion
}

#if MULTIPROTOCOL
static RAILSched_Config_t schedCfg; // Must never be const
static RAIL_Config_t railCfg = { // Must never be const
    .eventsCallback = &radioEventHandler,
    .protocol = NULL, // For BLE, pointer to a RAIL_BLE_State_t
    .scheduler = &schedCfg, // For MultiProtocol, additional scheduler memory
};
#else
static RAIL_Config_t railCfg = { // Must never be const
    .eventsCallback = &radioEventHandler,
    .protocol = NULL,
    .scheduler = NULL,
};
#endif

// Initializes the radio out of startup so that it's ready to receive.
void radioInitialize(void)
{
    // Initializes the RAIL library and any internal state it requires.
    gRailHandle = RAIL_Init(&railCfg, NULL);

    // Configures calibration settings.
    RAIL_ConfigCal(gRailHandle, RAIL_CAL_ALL);

    // Configures radio according to the generated radio settings.
    RAIL_ConfigChannels(gRailHandle, channelConfigs[0], &radioConfigChangedHandler);

    // Configures the most useful callbacks and catches a few errors.
    RAIL_ConfigEvents(gRailHandle,
        RAIL_EVENTS_ALL,
        RAIL_EVENT_TX_PACKET_SENT
        | RAIL_EVENT_RX_PACKET_RECEIVED
        | RAIL_EVENT_RX_FRAME_ERROR // invalid CRC
        | RAIL_EVENT_RX_ADDRESS_FILTERED);
}

```

```
// Sets automatic transitions to always receive once started.  
RAIL_StateTransitions_t railStateTransitions = {success = RAIL_RF_STATE_RX, error =  
RAIL_RF_STATE_RX};RAIL_SetRxTransitions(gRailHandle,&railStateTransitions);RAIL_SetTxTransitions(gRailHandle,&railStateTransitions);// Sets up the  
transmit buffer.RAIL_SetTxFifo(gRailHandle, txFifo,0, TX_FIFO_SIZE);}
```

Hardware-Specific Configuration

For hardware-specific configuration and setup information, see subsequent sections.

[EFR32](#)

RAIL API

RAIL API

This is the primary API layer for the Radio Abstraction Interface Layer (RAIL).

Modules

- [Antenna Control](#)
- [Assertions](#)
- [Auto-Ack](#)
- [Calibration](#)
- [Chip-Specific](#)
- [Data Management](#)
- [Diagnostic](#)
- [Events](#)
- [External Thermistor](#)
- [Features](#)
- [Front End Module Voltage Detection](#)
- [General](#)
- [Multiprotocol](#)
- [PTI Packet Trace](#)
- [Protocol-specific](#)
- [RF Sense](#)
- [RX Channel Hopping](#)
- [Radio Configuration](#)
- [Receive](#)
- [Retiming](#)
- [Sleep](#)
- [State Transitions](#)
- [System Timing](#)
- [TX Channel Hopping](#)
- [Thermal Protection](#)
- [Transmit](#)
- [TrustZone](#)

Antenna Control

Antenna Control

Basic APIs to control the antenna functionality.

These enumerations and structures are used with RAIL Antenna Control API.

EFR32 supports up to two antennas with configurable pin locations.

Modules

[RAIL_AntennaConfig_t](#)

Enumerations

```
enum RAIL_AntennaSel_t {
    RAIL_ANTENNA_0 = 0
    RAIL_ANTENNA_1 = 1
    RAIL_ANTENNA_AUTO = 255
}
```

Antenna path Selection enumeration.

Functions

- [RAIL_Status_t](#) [RAIL_ConfigAntenna](#)(RAIL_Handle_t railHandle, const RAIL_AntennaConfig_t *config)
Configure antenna path and pin locations.
- [RAIL_Status_t](#) [RAIL_GetRfPath](#)(RAIL_Handle_t railHandle, RAIL_AntennaSel_t *rfPath)
Get the default RF path.

Macros

```
#define ant0Loc defaultPath
```

Enumeration Documentation

RAIL_AntennaSel_t

RAIL_AntennaSel_t

Antenna path Selection enumeration.

Enumerator	
RAIL_ANTENNA_0	Enum for antenna path 0.
RAIL_ANTENNA_1	Enum for antenna path 1.
RAIL_ANTENNA_AUTO	Enum for antenna path auto.

Definition at line 4526 of file common/rail_types.h

Function Documentation

RAIL_ConfigAntenna

```
RAIL_Status_t RAIL_ConfigAntenna (RAIL_Handle_t railHandle, const RAIL_AntennaConfig_t * config)
```

Configure antenna path and pin locations.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.
[in]	config	A pointer to a configuration structure applied to the relevant Antenna Configuration registers. A NULL configuration will produce undefined behavior.

Warnings

- This API must be called before any TX or RX occurs. Otherwise, the antenna configurations for those functions will not take effect.

Returns

- Status code indicating success of the function call.

This function informs RAIL how to select each antenna, but not when. Antenna selection for receive is controlled by the [RAIL_RX_OPTION_ANTENNA0](#) and [RAIL_RX_OPTION_ANTENNA1](#) options (and the [RAIL_RX_OPTION_ANTENNA_AUTO](#) combination). Antenna selection for transmit is controlled by the [RAIL_TX_OPTION_ANTENNA0](#) and [RAIL_TX_OPTION_ANTENNA1](#) options.

There is only one antenna configuration can be active on a radio, regardless of the number of protocols (unless the application updates the configuration upon a protocol switch – RAIL does not save this configuration in a protocol RAIL instance).

Definition at line 533 of file common/rail.h

RAIL_GetRfPath

```
RAIL_Status_t RAIL_GetRfPath (RAIL_Handle_t railHandle, RAIL_AntennaSel_t * rfPath)
```

Get the default RF path.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	rfPath	A pointer to RF path updated by the function.

Returns

- Status code indicating success of the function call.

If multiple protocols are used, this function returns [RAIL_STATUS_INVALID_STATE](#) if it is called and the given railHandle is not active. In that case, the caller must attempt to re-call this function later, for example when [RAIL_EVENT_CONFIG_SCHEDULED](#) trigger.

Definition at line 548 of file common/rail.h

RAIL_AntennaConfig_t

A configuration for antenna selection.

Public Attributes

bool	ant0PinEn	Antenna 0 Pin Enable.
bool	ant1PinEn	Antenna 1 Pin Enable.
uint8_t	defaultPath	A RAIL_AntennaSel_t value specifying the internal default RF path.
uint8_t	ant0Port	Antenna 0 output GPIO port.
uint8_t	ant0Pin	Antenna 0 output GPIO pin.
uint8_t	ant1Loc	
uint8_t	ant1Port	Antenna 1 output GPIO port.
uint8_t	ant1Pin	Antenna 1 output GPIO pin.

Public Attribute Documentation

ant0PinEn

```
bool RAIL_AntennaConfig_t::ant0PinEn
```

Antenna 0 Pin Enable.

Definition at line 4548 of file common/rail_types.h

ant1PinEn

```
bool RAIL_AntennaConfig_t::ant1PinEn
```

Antenna 1 Pin Enable.

Definition at line 4550 of file common/rail_types.h

defaultPath

```
uint8_t RAIL_AntennaConfig_t::defaultPath
```

A [RAIL_AntennaSel_t](#) value specifying the internal default RF path.

It is ignored on EFR32 parts that have only one RF path bonded out and on EFR32xG28 dual-band OPNs where the appropriate RF path is automatically set by RAIL to 0 for 2.4 GHz band and 1 for Sub-GHz band PHYs. On EFR32xG23 and EFR32xG28 single-band OPNs where both RF paths are bonded out this can be set to [RAIL_ANTENNA_AUTO](#) to effect internal RF path diversity on PHYs supporting diversity. This avoids the need for an external RF switch and the associated GPIO(s) needed to control its antenna selection.

Definition at line 4564 of file common/rail_types.h

ant0Port

```
uint8_t RAIL_AntennaConfig_t::ant0Port
```

Antenna 0 output GPIO port.

Definition at line 4566 of file common/rail_types.h

ant0Pin

```
uint8_t RAIL_AntennaConfig_t::ant0Pin
```

Antenna 0 output GPIO pin.

Definition at line 4568 of file common/rail_types.h

ant1Loc

```
uint8_t RAIL_AntennaConfig_t::ant1Loc
```

DeprecatedNo longer used (ignored).

Definition at line 4570 of file common/rail_types.h

ant1Port

```
uint8_t RAIL_AntennaConfig_t::ant1Port
```

Antenna 1 output GPIO port.

Definition at line 4572 of file common/rail_types.h

ant1Pin

```
uint8_t RAIL_AntennaConfig_t::ant1Pin
```

Antenna 1 output GPIO pin.

Definition at line 4574 of file common/rail_types.h

Assertions

Assertions

Callbacks called by assertions.

The assertion framework was implemented to not only assert that certain conditions are true in a block of code, but also to handle them more appropriately. In previous implementations, the behavior upon a failed assert was to hang in a while(1) loop. However, with the callback, each assert is given a unique error code so that they can be handled on a more case-by-case basis. For documentation on each of the errors, see the [rail_assert_error_codes.h](#) file.

[RAIL_ASSERT_ERROR_MESSAGES](#)[errorCode] gives the explanation of the error. With asserts built into the library, users can choose how to handle each error inside the callback.

Enumerations

```
enum RAIL_AssertErrorCodes_t {
    RAIL_ASSERT_FAILED_APPENDED_INFO_MISSING = 0
    RAIL_ASSERT_FAILED_RX_FIFO_BYTES = 1
    RAIL_ASSERT_UNUSED_2 = 2
    RAIL_ASSERT_FAILED_ILLEGAL_RXLEN_ENTRY_STATUS = 3
    RAIL_ASSERT_FAILED_BAD_PACKET_LENGTH = 4
    RAIL_ASSERT_FAILED_SYNTH_DIVCTRL_ENUM_CONVERSION_ERROR = 5
    RAIL_ASSERT_FAILED_UNEXPECTED_STATE_RX_FIFO = 6
    RAIL_ASSERT_FAILED_UNEXPECTED_STATE_RXLEN_FIFO = 7
    RAIL_ASSERT_FAILED_UNEXPECTED_STATE_TX_FIFO = 8
    RAIL_ASSERT_FAILED_UNEXPECTED_STATE_TXACK_FIFO = 9
    RAIL_ASSERT_INVALID_MEMORY_ACCESS = 10
    RAIL_ASSERT_UNUSED_11 = 11
    RAIL_ASSERT_UNUSED_12 = 12
    RAIL_ASSERT_FAILED_RTCC_POST_WAKEUP = 13
    RAIL_ASSERT_FAILED_SYNTH_VCO_FREQUENCY = 14
    RAIL_ASSERT_FAILED_RAC_STATE = 15
    RAIL_ASSERT_FAILED_SYNTH_INVALID_VCOCTRL = 16
    RAIL_ASSERT_FAILED_NESTED_SEQUENCER_LOCK = 17
    RAIL_ASSERT_FAILED_RSSI_AVERAGE_DONE = 18
    RAIL_ASSERT_UNUSED_19 = 19
    RAIL_ASSERT_FAILED_PROTIMER_RANDOM_SEED = 20
    RAIL_ASSERT_UNUSED_21 = 21
    RAIL_ASSERT_FAILED_PROTIMER_CHANNEL = 22
    RAIL_ASSERT_UNUSED_23 = 23
    RAIL_ASSERT_FAILED_BASECNTTOP = 24
    RAIL_ASSERT_UNUSED_25 = 25
    RAIL_ASSERT_FAILED_RTCC_SYNC_MISSED = 26
    RAIL_ASSERT_FAILED_CLOCK_SOURCE_NOT_READY = 27
    RAIL_ASSERT_UNUSED_28 = 28
    RAIL_ASSERT_NULL_HANDLE = 29
    RAIL_ASSERT_UNUSED_30 = 30
    RAIL_ASSERT_FAILED_NO_ACTIVE_CONFIG = 31
    RAIL_ASSERT_UNUSED_32 = 32
    RAIL_ASSERT_UNUSED_33 = 33
    RAIL_ASSERT_UNUSED_34 = 34
    RAIL_ASSERT_UNUSED_35 = 35
    RAIL_ASSERT_UNUSED_36 = 36
    RAIL_ASSERT_UNUSED_37 = 37
    RAIL_ASSERT_FAILED_TX_SYNTH_ENABLE = 38
    RAIL_ASSERT_DEPRECATED_FUNCTION = 39
    RAIL_ASSERT_MULTIPROTOCOL_NO_EVENT = 40
    RAIL_ASSERT_FAILED_INVALID_INTERRUPT_ENABLED = 41
    RAIL_ASSERT_UNUSED_42 = 42
    RAIL_ASSERT_DIVISION_BY_ZERO = 43
    RAIL_ASSERT_CANT_USE_HARDWARE = 44
    RAIL_ASSERT_NULL_PARAMETER = 45
    RAIL_ASSERT_SECURE_ELEMENT_FAULT = 46
    RAIL_ASSERT_SMALL_SYNTH_RADIO_CONFIG_BUFFER = 47
    RAIL_ASSERT_CHANNEL_HOPPING_BUFFER_TOO_SHORT = 48
    RAIL_ASSERT_INVALID_MODULE_ACTION = 49
    RAIL_ASSERT_CHANNEL_HOPPING_INVALID_RADIO_CONFIG = 50
    RAIL_ASSERT_CHANNEL_CHANGE_FAILED = 51
    RAIL_ASSERT_INVALID_REGISTER = 52
    RAIL_ASSERT_CP_DMA_INTERNAL_GENERIC_ERROR = 53
    RAIL_ASSERT_CACHE_CONFIG_FAILED = 54
    RAIL_ASSERT_NULL_TRANSITIONS = 55
    RAIL_ASSERT_BAD_LDMA_TRANSFER = 56
    RAIL_ASSERT_INVALID_RTCC_SYNC_VALUES = 57
    RAIL_ASSERT_SEQUENCER_FAULT = 58
    RAIL_ASSERT_BUS_ERROR = 59
    RAIL_ASSERT_INVALID_FILTERING_CONFIG = 60
    RAIL_ASSERT_RETIMING_CONFIG = 61
    RAIL_ASSERT_FAILED_TX_CRC_CONFIG = 62
    RAIL_ASSERT_INVALID_PA_OPERATION = 63
    RAIL_ASSERT_SEQ_INVALID_PA_SELECTED = 64
    RAIL_ASSERT_FAILED_INVALID_CHANNEL_CONFIG = 65
    RAIL_ASSERT_INVALID_XTAL_FREQUENCY = 66
    RAIL_ASSERT_INTERNAL_GENERIC_ERROR = 67
    RAIL_ASSERT_UNSUPPORTED_SOFTWARE_MODEM_MODULATION = 68
}
```

```
RAIL_ASSERT_FAILED_RTCC_SYNC_STOP = 69
RAIL_ASSERT_FAILED_MULTITIMER_CORRUPT = 70
RAIL_ASSERT_FAILED_TEMPCAL_ERROR = 71
RAIL_ASSERT_INVALID_VDET_CONFIGURATION = 72
RAIL_ASSERT_INVALID_RFFPLL_CONFIGURATION = 73
RAIL_ASSERT_SECURE_ACCESS_FAULT = 74
RAIL_ASSERT_FAILED_SYSRTC0_NOT_RUNNING = 75
RAIL_ASSERT_RADIO_CONFIG_NOT_UP_TO_DATE = 76
RAIL_ASSERT_FAILED_RSSI_THRESHOLD = 77
RAIL_ASSERT_INCORRECT_ZWAVE_REGION = 78
RAIL_ASSERT_FAILED_RTCC_SYNC_STALE_DATA = 79
RAIL_ASSERT_INVALID_LOG2X4_CLEAR_CONDITION = 80
RAIL_ASSERT_FAILED_DMA_WRITE_INCOMPLETE = 81
RAIL_ASSERT_CALCULATOR_NOT_SUPPORTED = 82
RAIL_ASSERT_INVALID_SEQUENCER_IMAGE = 83
RAIL_ASSERT_MISSING_SEQUENCER_IMAGE = 84
RAIL_ASSERT_INVALID_OR_MISSING_SOFTWARE_MODEM_IMAGE
= 85
RAIL_ASSERT_SEQ_USER_SEQUENCER_GENERIC_ERROR = 86
}
```

Enumeration of all possible error codes from RAIL_ASSERT.

Functions

```
void RAILCb_AssertFailed(RAIL_Handle_t railHandle, RAIL_AssertErrorCodes_t errorCode)
    Callback called upon failed assertion.
```

Macros

```
#define RAIL_ASSERT_ERROR_MESSAGES
    Use this define to create an array of error messages that map to the codes in RAIL_AssertErrorCodes_t.
```

Enumeration Documentation

RAIL_AssertErrorCodes_t

RAIL_AssertErrorCodes_t

Enumeration of all possible error codes from RAIL_ASSERT.

Enumerator	
RAIL_ASSERT_FAILED_APPENDED_INFO_MISSING	Appended info missing from RX packet.
RAIL_ASSERT_FAILED_RX_FIFO_BYTES	Receive FIFO too small for IR calibration.
RAIL_ASSERT_UNUSED_2	Invalid assert, no longer used.
RAIL_ASSERT_FAILED_ILLEGAL_RXLEN_ENTRY_STATUS	Receive FIFO entry has invalid status.
RAIL_ASSERT_FAILED_BAD_PACKET_LENGTH	Receive FIFO entry bad packet length.
RAIL_ASSERT_FAILED_SYNTH_DIVCTRL_ENUM_CONVERSION_ERROR	Unable to configure radio for IR calibration.
RAIL_ASSERT_FAILED_UNEXPECTED_STATE_RX_FIFO	Reached unexpected state while handling RX FIFO events.
RAIL_ASSERT_FAILED_UNEXPECTED_STATE_RXLEN_FIFO	Reached unexpected state while handling RXLEN FIFO events.
RAIL_ASSERT_FAILED_UNEXPECTED_STATE_TX_FIFO	Reached unexpected state while handling TX FIFO events.
RAIL_ASSERT_FAILED_UNEXPECTED_STATE_TXACK_FIFO	Reached unexpected state while handling TX ACK FIFO events.

RAIL_ASSERT_INVALID_MEMORY_ACCESS	Invalid memory region accessed.
RAIL_ASSERT_UNUSED_11	Invalid assert, no longer used.
RAIL_ASSERT_UNUSED_12	Invalid assert, no longer used.
RAIL_ASSERT_FAILED_RTCC_POST_WAKEUP	Error synchronizing the RAIL timebase after sleep.
RAIL_ASSERT_FAILED_SYNTH_VCO_FREQUENCY	VCO frequency outside supported range.
RAIL_ASSERT_FAILED_RAC_STATE	Radio active while changing channels.
RAIL_ASSERT_FAILED_SYNTH_INVALID_VCOCTRL	Invalid Synth VCOCTRL field calculation.
RAIL_ASSERT_FAILED_NESTED_SEQUENCER_LOCK	Nested attempt to lock the sequencer.
RAIL_ASSERT_FAILED_RSSI_AVERAGE_DONE	RSSI averaging enabled without a valid callback.
RAIL_ASSERT_UNUSED_19	Invalid assert, no longer used.
RAIL_ASSERT_FAILED_PROTIMER_RANDOM_SEED	Unable to seed radio pseudo random number generator.
RAIL_ASSERT_UNUSED_21	Invalid assert, no longer used.
RAIL_ASSERT_FAILED_PROTIMER_CHANNEL	Invalid timer channel specified.
RAIL_ASSERT_UNUSED_23	Invalid assert, no longer used.
RAIL_ASSERT_FAILED_BASECNTTOP	LBT config exceeds register size.
RAIL_ASSERT_UNUSED_25	Invalid assert, no longer used.
RAIL_ASSERT_FAILED_RTCC_SYNC_MISSED	Could not synchronize RAIL timebase with the RTC.
RAIL_ASSERT_FAILED_CLOCK_SOURCE_NOT_READY	Clock source not ready.
RAIL_ASSERT_UNUSED_28	Invalid assert, no longer used.
RAIL_ASSERT_NULL_HANDLE	NULL was supplied as a RAIL_Handle_t argument.
RAIL_ASSERT_UNUSED_30	Invalid assert, no longer used.
RAIL_ASSERT_FAILED_NO_ACTIVE_CONFIG	API improperly called while protocol inactive.
RAIL_ASSERT_UNUSED_32	Invalid assert, no longer used.
RAIL_ASSERT_UNUSED_33	Invalid assert, no longer used.
RAIL_ASSERT_UNUSED_34	Invalid assert, no longer used.
RAIL_ASSERT_UNUSED_35	Invalid assert, no longer used.
RAIL_ASSERT_UNUSED_36	Invalid assert, no longer used.
RAIL_ASSERT_UNUSED_37	Invalid assert, no longer used.
RAIL_ASSERT_FAILED_TX_SYNTH_ENABLE	Failed to enable synth for transmit.
RAIL_ASSERT_DEPRECATED_FUNCTION	This function is deprecated and must not be called.
RAIL_ASSERT_MULTIPROTOCOL_NO_EVENT	Multiprotocol task started with no event to run.
RAIL_ASSERT_FAILED_INVALID_INTERRUPT_ENABLED	Invalid interrupt enabled.
RAIL_ASSERT_UNUSED_42	Invalid assert, no longer used.
RAIL_ASSERT_DIVISION_BY_ZERO	Division by zero.
RAIL_ASSERT_CANT_USE_HARDWARE	Function cannot be called without access to the hardware.
RAIL_ASSERT_NULL_PARAMETER	Pointer parameter was passed as NULL.
RAIL_ASSERT_SECURE_ELEMENT_FAULT	Secure Element fault.
RAIL_ASSERT_SMALL_SYNTH_RADIO_CONFIG_BUFFER	Synth radio config buffer for channel hopping too small.
RAIL_ASSERT_CHANNEL_HOPPING_BUFFER_TOO_SHORT	Buffer provided for RX Channel Hopping is too small.
RAIL_ASSERT_INVALID_MODULE_ACTION	Invalid action was attempted on a module.
RAIL_ASSERT_CHANNEL_HOPPING_INVALID_RADIO_CONFIG	The radio config for this channel is not compatible with channel hopping.
RAIL_ASSERT_CHANNEL_CHANGE_FAILED	Channel change failed.

RAIL_ASSERT_INVALID_REGISTER	Attempted to read invalid register.
RAIL_ASSERT_CP_DMA_INTERNAL_GENERIC_ERROR	CP/DMA Invalid error.
RAIL_ASSERT_CACHE_CONFIG_FAILED	DMP radio config caching failed.
RAIL_ASSERT_NULL_TRANSITIONS	NULL was supplied as a RAIL_StateTransitions_t argument.
RAIL_ASSERT_BAD_LDMA_TRANSFER	LDMA transfer failed.
RAIL_ASSERT_INVALID_RTCC_SYNC_VALUES	Attempted to wake up with invalid RTCC sync data.
RAIL_ASSERT_SEQUENCER_FAULT	Radio sequencer hit a fault condition.
RAIL_ASSERT_BUS_ERROR	Bus fault.
RAIL_ASSERT_INVALID_FILTERING_CONFIG	The current radio config cannot be used with packet filtering.
RAIL_ASSERT_RETIMING_CONFIG	Retiming configuration error.
RAIL_ASSERT_FAILED_TX_CRC_CONFIG	TX CRC configuration is corrupt.
RAIL_ASSERT_INVALID_PA_OPERATION	The current PA config does not allow for this operation.
RAIL_ASSERT_SEQ_INVALID_PA_SELECTED	The sequencer selected an invalid PA.
RAIL_ASSERT_FAILED_INVALID_CHANNEL_CONFIG	Invalid/unsupported channel config.
RAIL_ASSERT_INVALID_XTAL_FREQUENCY	Radio Calculator configuration HFXO frequency mismatch with chip.
RAIL_ASSERT_INTERNAL_GENERIC_ERROR	Internal error.
RAIL_ASSERT_UNSUPPORTED_SOFTWARE_MODEM_MODULATION	Software modem image does not support requested modulation
RAIL_ASSERT_FAILED_RTCC_SYNC_STOP	Failed to disable RTCC synchronization.
RAIL_ASSERT_FAILED_MULTITIMER_CORRUPT	Multitimer linked list corrupted.
RAIL_ASSERT_FAILED_TEMPCAL_ERROR	Unable to configure radio for temperature calibration.
RAIL_ASSERT_INVALID_VDET_CONFIGURATION	Invalid VDET configuration.
RAIL_ASSERT_INVALID_RFFPLL_CONFIGURATION	Invalid RFFPLL configuration.
RAIL_ASSERT_SECURE_ACCESS_FAULT	Secure access fault.
RAIL_ASSERT_FAILED_SYSRTC0_NOT_RUNNING	SYSRTC0 not running.
RAIL_ASSERT_RADIO_CONFIG_NOT_UP_TO_DATE	Radio Configurator not updated.
RAIL_ASSERT_FAILED_RSSI_THRESHOLD	Failed to set the event for configurable RSSI threshold.
RAIL_ASSERT_INCORRECT_ZWAVE_REGION	Intended and actual Z-Wave region configuration mismatch.
RAIL_ASSERT_FAILED_RTCC_SYNC_STALE_DATA	Attempted to sleep with stale RTCC synchronization data.
RAIL_ASSERT_INVALID_LOG2X4_CLEAR_CONDITION	Attempted to clear LOG2X4 with a DEC1 value not equal to 0.
RAIL_ASSERT_FAILED_DMA_WRITE_INCOMPLETE	Failed to complete DMA write.
RAIL_ASSERT_CALCULATOR_NOT_SUPPORTED	RAIL does not support this Radio Calculator configuration.
RAIL_ASSERT_INVALID_SEQUENCER_IMAGE	Invalid binary image was loaded onto the sequencer.
RAIL_ASSERT_MISSING_SEQUENCER_IMAGE	No common or protocol image selected to be loaded onto the sequencer.
RAIL_ASSERT_INVALID_OR_MISSING_SOFTWARE_MODEM_IMAGE	Software modem image invalid or missing.
RAIL_ASSERT_SEQ_USER_SEQUENCER_GENERIC_ERROR	The sequencer user generated error.

Definition at line 50 of file common/rail_assert_error_codes.h

Function Documentation

RAILCb_AssertFailed

```
void RAILCb_AssertFailed (RAIL_Handle_t railHandle, RAIL_AssertErrorCodes_t errorCode)
```

Callback called upon failed assertion.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	errorCode	Value passed in by the calling assertion API indicating the RAIL assertion that failed.

Definition at line 6686 of file common/rail.h

Auto-Ack

Auto-Ack

APIs for configuring Auto-Ack functionality.

These APIs configure the radio for automatic acknowledgment features. Auto-Ack inherently changes how the underlying state machine behaves so users should not modify [RAIL_SetRxTransitions\(\)](#) and [RAIL_SetTxTransitions\(\)](#) while using Auto-Ack features.

```
// Go to RX after Ack operation.
RAIL_AutoAckConfig_t autoAckConfig = {
    .enable = true,
    .ackTimeout = 1000,
    .rxTransitions = {
        .success = RAIL_RF_STATE_RX,
        .error = RAIL_RF_STATE_RX
    },
    .txTransitions = {
        .success = RAIL_RF_STATE_RX,
        .error = RAIL_RF_STATE_RX,
    },
};

RAIL_Status_t status = RAIL_ConfigAutoAck(railHandle, &autoAckConfig);

uint8_t ackData[] = {0x05, 0x02, 0x10, 0x00};

RAIL_Status_t status = RAIL_WriteAutoAckFifo(railHandle,
    ackData,
    sizeof(ackData));
```

The acknowledgment transmits based on the frame format configured via the Radio Configurator. For example, if the frame format is using a variable length scheme, the Ack will be sent according to that scheme. If a 10-byte packet is loaded into the Ack, but the variable length field of the Ack payload specifies a length of 5, only 5 bytes will transmit for the Ack. The converse is also true, if the frame length is configured to be a fixed 10-byte packet but only 5 bytes are loaded into the Ack buffer, a TX underflow occurs during the Ack transmit.

Unlike in non-Auto-Ack mode, Auto-Ack mode will always return to a single state after all Ack sequences complete, regardless of whether the Ack was successfully received/sent or not. See the documentation of [RAIL_ConfigAutoAck\(\)](#) for configuration information. To suspend automatic acknowledgment of a series of packets after transmit or receive call [RAIL_PauseTxAutoAck\(\)](#) or [RAIL_PauseRxAutoAck\(\)](#) respectively with the pause parameter set to true. When Auto-Acking is paused, after receiving or transmitting a packet (regardless of success), the radio transitions to the same state it would use while Acking. To return to normal state transition logic outside of Acking, call [RAIL_ConfigAutoAck\(\)](#) with the [RAIL_AutoAckConfig_t::enable](#) field false and specify the desired transitions in the [RAIL_AutoAckConfig_t::rxTransitions](#) and [RAIL_AutoAckConfig_t::txTransitions](#) fields. To get out of a paused state and resume Auto-Acking, call [RAIL_PauseTxAutoAck\(\)](#) and/or [RAIL_PauseRxAutoAck\(\)](#) with the pause parameter set to false.

Applications can cancel the transmission of an Ack with [RAIL_CancelAutoAck\(\)](#). Conversely, applications can control if a transmit operation should wait for an Ack after transmitting by using the [RAIL_TX_OPTION_WAIT_FOR_ACK](#) option.

When [Antenna Control](#) is used for multiple antennas, Acks are transmitted on the antenna that was selected to receive the packet being acknowledged. When receiving an Ack, the [RAIL_RxOptions_t](#) antenna options are used just like for any other receive.

If the Ack payload is dynamic, the application must call `RAIL_WriteAutoAckFifo()` with the appropriate Ack payload after the application processes the receive. RAIL can Auto-Ack from the normal transmit buffer if `RAIL_UseTxFifoForAutoAck()` is called before the radio transmits the Ack. Ensure the transmit buffer contains data loaded by `RAIL_WriteTxFifo()`.

Standard-based protocols that contain Auto-Ack functionality are normally configured in the protocol-specific configuration function. For example, `RAIL_IEEE802154_Init()` provides Auto-Ack configuration parameters in `RAIL_IEEE802154_Config_t` and should only be configured through that function. It is not advisable to call both `RAIL_IEEE802154_Init()` and `RAIL_ConfigAutoAck()`. However, Ack modification functions are still valid to use with protocol-specific Acks. To cancel an IEEE 802.15.4 Ack transmit, use `RAIL_CancelAutoAck()`.

Modules

`RAIL_AutoAckConfig_t`

Functions

<code>RAIL_Status_t</code>	<code>RAIL_ConfigAutoAck(RAIL_Handle_t railHandle, const RAIL_AutoAckConfig_t *config)</code> Configure and enable automatic acknowledgment.
<code>bool</code>	<code>RAIL_IsAutoAckEnabled(RAIL_Handle_t railHandle)</code> Return the enable status of the Auto-Ack feature.
<code>RAIL_Status_t</code>	<code>RAIL_WriteAutoAckFifo(RAIL_Handle_t railHandle, const uint8_t *ackData, uint16_t ackDataLen)</code> Load the Auto-Ack buffer with Ack data.
<code>RAIL_Status_t</code>	<code>RAIL_GetAutoAckFifo(RAIL_Handle_t railHandle, uint8_t **ackBuffer, uint16_t *ackBufferBytes)</code> Get the address and size of the Auto-Ack transmit buffer for direct access.
<code>RAIL_Status_t</code>	<code>RAIL_PauseRxAutoAck(RAIL_Handle_t railHandle, bool pause)</code> Pause/resume RX Auto-Ack functionality.
<code>bool</code>	<code>RAIL_IsRxAutoAckPaused(RAIL_Handle_t railHandle)</code> Return whether the RX Auto-Ack is paused.
<code>RAIL_Status_t</code>	<code>RAIL_PauseTxAutoAck(RAIL_Handle_t railHandle, bool pause)</code> Pause/resume TX Auto-Ack functionality.
<code>bool</code>	<code>RAIL_IsTxAutoAckPaused(RAIL_Handle_t railHandle)</code> Return whether the TX Auto-Ack is paused.
<code>RAIL_Status_t</code>	<code>RAIL_UseTxFifoForAutoAck(RAIL_Handle_t railHandle)</code> Modify the upcoming Ack to use the transmit FIFO.
<code>RAIL_Status_t</code>	<code>RAIL_CancelAutoAck(RAIL_Handle_t railHandle)</code> Cancel the upcoming Ack.
<code>bool</code>	<code>RAIL_IsAutoAckWaitingForAck(RAIL_Handle_t railHandle)</code> Return whether the radio is currently waiting for an Ack.

Macros

<code>#define</code>	<code>RAIL_AUTOACK_MAX_LENGTH (64U)</code> Acknowledgment packets cannot be longer than 64 bytes.

Function Documentation

`RAIL_ConfigAutoAck`

`RAIL_Status_t RAIL_ConfigAutoAck (RAIL_Handle_t railHandle, const RAIL_AutoAckConfig_t * config)`

Configure and enable automatic acknowledgment.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	A pointer to an Auto-Ack configuration structure.

Returns

- Status code indicating success of the function call.

Configures the RAIL state machine for hardware-accelerated automatic acknowledgment. Ack timing parameters are defined in the configuration structure.

While Auto-Acking is enabled, do not call the following RAIL functions:

- [RAIL_SetRxTransitions\(\)](#)
- [RAIL_SetTxTransitions\(\)](#)

When Auto-Acking is enabled, only one state transition can be defined (without notion of success or error). Thus if you are enabling Auto-Ack (i.e., config->enable field is true) the "error" states of config->rxTransitions and config->txTransitions are ignored. After all Ack sequences, (success or fail) the state machine will return the radio to the "success" state, which can be either [RAIL_RF_STATE_RX](#) or [RAIL_RF_STATE_IDLE](#) (returning to [RAIL_RF_STATE_TX](#) is not supported). Alternatively when Auto-Acking is disabled (i.e., config->enable field is false), transitions are reconfigured using all fields of config->rxTransitions and config->txTransitions. When disabling, the "ackTimeout" field isn't used.

If you need information about the actual success of the Ack sequence, use RAIL events such as [RAIL_EVENT_TXACK_PACKET_SENT](#) to make sure an Ack was sent, or [RAIL_EVENT_RX_ACK_TIMEOUT](#) to make sure that an Ack was received within the specified timeout.

To set a certain turnaround time (i.e., txToRx and rxToTx in [RAIL_StateTiming_t](#)), make txToRx lower than desired to ensure you get to RX in time to receive the Ack. Silicon Labs recommends setting 10 us lower than desired:

```
void setAutoAckStateTimings(void)
{
    RAIL_StateTiming_t timings;

    // User is already in Auto-Ack and wants a turnaround of 192 us.
    timings.rxToTx = 192;
    timings.txToRx = 192 - 10;

    // Set other fields of timings...
    timings.idleToRx = 100;
    timings.idleToTx = 100;
    timings.rxSearchTimeout = 0;
    timings.txToRxSearchTimeout = 0;

    RAIL_SetStateTiming(railHandle, &timings);
}
```

Note

- Auto-Acking may not be enabled while RX Channel Hopping is enabled, or when BLE is enabled.

Definition at line 4947 of file `common/rail.h`

RAIL_IsAutoAckEnabled

```
bool RAIL_IsAutoAckEnabled (RAIL_Handle_t railHandle)
```

Return the enable status of the Auto-Ack feature.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if Auto-Ack is enabled, false if disabled.

Definition at line 4956 of file common/rail.h

RAIL_WriteAutoAckFifo

```
RAIL_Status_t RAIL_WriteAutoAckFifo (RAIL_Handle_t railHandle, const uint8_t * ackData, uint16_t ackDataLen)
```

Load the Auto-Ack buffer with Ack data.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	ackData	A pointer to Ack data to transmit. This may be NULL, in which case it's assumed the data has already been emplaced into the Ack buffer and RAIL just needs to be told how many bytes are there. Use RAIL_GetAutoAckFifo() to get the address of RAIL's Auto-Ack buffer in RAM and its size.
[in]	ackDataLen	The number of bytes in Ack data.

Returns

- Status code indicating success of the function call.

If the Ack buffer is available for updates, load the Ack buffer with data. If it is not available, [RAIL_STATUS_INVALID_STATE](#) is returned. If ackDataLen exceeds [RAIL_AUTOACK_MAX_LENGTH](#) then [RAIL_STATUS_INVALID_PARAMETER](#) will be returned and nothing is written to the Ack buffer (unless ackData is NULL in which case this indicates the application has already likely corrupted RAM).

Definition at line 4977 of file common/rail.h

RAIL_GetAutoAckFifo

```
RAIL_Status_t RAIL_GetAutoAckFifo (RAIL_Handle_t railHandle, uint8_t ** ackBuffer, uint16_t * ackBufferBytes)
```

Get the address and size of the Auto-Ack transmit buffer for direct access.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	ackBuffer	A pointer to a uint8_t pointer that will be updated to the RAM base address of the Auto-Ack FIFO buffer.
[inout]	ackBufferBytes	A pointer to a uint16_t that will be updated to the size of the Auto-Ack FIFO buffer, in bytes, currently RAIL_AUTOACK_MAX_LENGTH .

Returns

- Status code indicating success of the function call.

Applications can use this to more flexibly write Auto-Ack data into the buffer directly and in pieces, passing NULL ackData parameter to [RAIL_WriteAutoAckFifo\(\)](#) or [RAIL_IEEE802154_WriteEnhAck\(\)](#) to inform RAIL of its final length.

Definition at line 4997 of file common/rail.h

RAIL_PauseRxAutoAck

```
RAIL_Status_t RAIL_PauseRxAutoAck (RAIL_Handle_t railHandle, bool pause)
```

Pause/resume RX Auto-Ack functionality.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	pause	Pause or resume RX Auto-Acking.

Returns

- Status code indicating success of the function call.
- When RX Auto-Acking is paused, the radio transitions to [RAIL_AutoAckConfig_t::rxTransitions's RAIL_StateTransitions_t::success](#) state after receiving a packet and does not transmit an Ack. When RX Auto-Ack is resumed, the radio resumes automatically Acking every successfully received packet.

Definition at line 5014 of file common/rail.h

RAIL_IsRxAutoAckPaused

```
bool RAIL_IsRxAutoAckPaused (RAIL_Handle_t railHandle)
```

Return whether the RX Auto-Ack is paused.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if RX Auto-Ack is paused, false if not paused.

Definition at line 5023 of file common/rail.h

RAIL_PauseTxAutoAck

```
RAIL_Status_t RAIL_PauseTxAutoAck (RAIL_Handle_t railHandle, bool pause)
```

Pause/resume TX Auto-Ack functionality.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	pause	Pause or resume TX Auto-Acking.

Returns

- Status code indicating success of the function call.
- When TX Auto-Acking is paused, the radio transitions to [RAIL_AutoAckConfig_t::txTransitions's RAIL_StateTransitions_t::success](#) state after transmitting a packet and does not wait for an Ack. When TX Auto-Ack is resumed, the radio resumes automatically waiting for an Ack after a successful transmit.

Definition at line 5038 of file common/rail.h

RAIL_IsTxAutoAckPaused

bool RAIL_IsTxAutoAckPaused (RAIL_Handle_t railHandle)

Return whether the TX Auto-Ack is paused.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if TX Auto-Ack is paused, false if not paused.

Definition at line 5046 of file common/rail.h

RAIL_UseTxFifoForAutoAck

RAIL_Status_t RAIL_UseTxFifoForAutoAck (RAIL_Handle_t railHandle)

Modify the upcoming Ack to use the transmit FIFO.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call. The call will fail if it is too late to modify the outgoing Ack.

This function allows the application to use the normal transmit FIFO as the data source for the upcoming Ack. The Ack modification to use the transmit FIFO only applies to one Ack transmission.

This function only returns true if the following conditions are met:

- Radio has not already decided to use the Ack buffer AND
- Radio is either looking for sync, receiving the packet after sync, or in the [RAIL_StateTiming_t::rxToTx](#) turnaround before the Ack is sent.

Note

- The transmit FIFO must not be used for Auto-Ack when IEEE 802.15.4, Z-Wave, or BLE protocols are active.

Definition at line 5067 of file common/rail.h

RAIL_CancelAutoAck

RAIL_Status_t RAIL_CancelAutoAck (RAIL_Handle_t railHandle)

Cancel the upcoming Ack.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

Status code indicating success of the function call. This call will fail if it is too late to modify the outgoing Ack.

This function allows the application to cancel the upcoming automatic acknowledgment.

This function only returns true if the following conditions are met:

- Radio has not already decided to transmit the Ack, and
- Radio is either looking for sync, receiving the packet after sync or in the [RAIL_StateTiming_t::rxToTx](#) turnaround before the Ack is sent.

Definition at line 5084 of file `common/rail.h`

RAIL_IsAutoAckWaitingForAck

```
bool RAIL_IsAutoAckWaitingForAck (RAIL\_Handle\_t railHandle)
```

Return whether the radio is currently waiting for an Ack.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if radio is waiting for Ack, false if radio is not waiting for an Ack.

This function allows the application to query whether the radio is currently waiting for an Ack after a transmit operation.

Definition at line 5096 of file `common/rail.h`

RAIL_AutoAckConfig_t

Enable/disable the Auto-Ack algorithm, based on "enable".

The structure provides a default state (the "success" of TX/RX transitions when Acking is enabled) for the radio to return to after an Ack operation occurs (transmitting or attempting to receive an Ack), or normal state transitions to return to in the case Acking is disabled. Regardless whether the Ack operation was successful, the radio returns to the specified success state.

ackTimeout specifies how long to stay in receive and wait for an Ack to start (sync detected) before issuing a [RAIL_EVENT_RX_ACK_TIMEOUT](#) event and return to the default state.

Public Attributes

<code>bool</code>	enable	Indicate whether Auto-Acking should be enabled or disabled.
<code>uint16_t</code>	ackTimeout	Define the RX Ack timeout duration in microseconds up to 65535 microseconds maximum.
RAIL_StateTransitions_t	rxTransitions	State transitions to do after receiving a packet.
RAIL_StateTransitions_t	txTransitions	State transitions to do after transmitting a packet.

Public Attribute Documentation

enable

```
bool RAIL_AutoAckConfig_t::enable
```

Indicate whether Auto-Acking should be enabled or disabled.

Definition at line `4475` of file `common/rail_types.h`

ackTimeout

```
uint16_t RAIL_AutoAckConfig_t::ackTimeout
```

Define the RX Ack timeout duration in microseconds up to 65535 microseconds maximum.

Only applied when Auto-Acking is enabled. The Ack timeout timer starts at the completion of a [RAIL_TX_OPTION_WAIT_FOR_ACK](#) transmit and expires only while waiting for a packet (prior to SYNC detect), triggering [RAIL_EVENT_RX_ACK_TIMEOUT](#). During packet reception that event is held off until packet completion and suppressed entirely if the received packet is the expected Ack.

Definition at line `4487` of file `common/rail_types.h`

rxTransitions

```
RAIL_StateTransitions_t RAIL_AutoAckConfig_t::rxTransitions
```

State transitions to do after receiving a packet.

When Auto-Acking is enabled, the "error" transition is always ignored and the radio will return to the "success" state after any Acking sequence ([RAIL_RF_STATE_RX](#) or [RAIL_RF_STATE_IDLE](#)). See [RAIL_ConfigAutoAck\(\)](#) for more details on this.

Definition at line 4495 of file `common/rail_types.h`

txTransitions

```
RAIL_StateTransitions_t RAIL_AutoAckConfig_t::txTransitions
```

State transitions to do after transmitting a packet.

When Auto-Acking is enabled, the "error" transition is always ignored and the radio will return to the "success" state after any Acking sequence ([RAIL_RF_STATE_RX](#) or [RAIL_RF_STATE_IDLE](#)). See [RAIL_ConfigAutoAck\(\)](#) for more details on this.

Definition at line 4503 of file `common/rail_types.h`

Calibration

Calibration

APIs for calibrating the radio.

IEEE802154 protocol-specific APIs for calibrating the radio.

Bluetooth protocol-specific APIs for calibrating the radio.

The EFR32 supports the Image Rejection (IR) calibration and a temperature-dependent calibration.

These APIs calibrate the radio. The RAIL library determines which calibrations are necessary. Calibrations can be enabled/disabled with the [RAIL_CalMask_t](#) parameter.

Some calibrations produce values that can be saved and reapplied to avoid repeating the calibration process.

Calibrations can either be run with [RAIL_Calibrate\(\)](#), or with the individual chip-specific calibration routines. An example for running code with [RAIL_Calibrate\(\)](#) looks like the following:

```
static RAIL_CalValues_t calValues = RAIL_CALVALUES_UNINIT;
static volatile bool calibrateRadio = false;

void RAILCb_Event(RAIL_Handle_t railHandle, RAIL_Events_t events)
{
    // Omitting other event handlers
    if (events & RAIL_EVENT_CAL_NEEDED) {
        calibrateRadio = true;
    }
}

void main(void)
{
    // Initialize RAIL ...

    // Application main loop
    while (1) {
        ...
        if (calibrateRadio) {
            // Run all pending calibrations, and save the results
            RAIL_Calibrate(railHandle, &calValues, RAIL_CAL_ALL_PENDING);
            calibrateRadio = false;
        }
        ...
    }
}
```

Alternatively, if the image rejection calibration for your radio can be determined ahead of time, such as by running the calibration on a separate firmware image on each chip, the following calibration process will result in smaller code.

```
static RAIL_IrCalValues_t irCalValues = {
    .rxIrCalValues = {
        RX_IRCAL_VALUE_RF_PATH0,
        RX_IRCAL_VALUE_RF_PATH1,
    },
    .txIrCalValues = {
        dcOffsetIQ = TX_IRCAL_DC_OFFSET_IQ,
        phiEpsilon = TX_IRCAL_PHLEPSILON,
    },
};

void RAILCb_Event(RAIL_Handle_t railHandle, RAIL_Events_t events)
{
    // Omitting other event handlers
    if (events & RAIL_EVENT_CAL_NEEDED) {
        RAIL_CalMask_t pendingCals = RAIL_GetPendingCal(railHandle);
        // Disable the radio if we have to do an offline calibration
        if (pendingCals & RAIL_CAL_TEMP_VCO) {
            (void) RAIL_CalibrateTemp(railHandle);
        }
        if (pendingCals & RAIL_CAL_ONETIME_IRCAL) {
            (void) RAIL_ApplyIrCalibrationAlt(railHandle, &irCalValues, RAIL_ANTENNA_AUTO);
        }
    }
}
```

The IR calibration can be computed once and stored off or computed each time at startup. Because it is PHY-specific and provides sensitivity improvements, it is highly recommended. The IR calibration should only be run when the radio is IDLE.

The temperature-dependent calibrations are used to recalibrate the synth if the temperature crosses 0C or the temperature delta since the last calibration exceeds 70C while in receive. RAIL will run the VCO calibration automatically upon entering receive or transmit states, so the application can omit this calibration if the stack re-enters receive or transmit with enough frequency to avoid reaching the temperature delta. If the application does not calibrate for temperature, it's possible to miss receive packets due to a drift in the carrier frequency.

Modules

[RAIL_TxIrCalValues_t](#)

[RAIL_IrCalValues_t](#)

[EFR32xG2x](#)

[Slxx3xx](#)

Typedefs

typedef uint32_t [RAIL_CalMask_t](#)
A calibration mask type.

typedef uint32_t [RAIL_RxIrCalValues_t](#)[2]
[RAIL_RxIrCalValues_t](#).

typedef [RAIL_CalValues_t](#)
[RAIL_IrCalValues_t](#) A calibration value structure.

Functions

[RAIL_Status_t](#) [RAIL_ConfigCal](#)(RAIL_Handle_t railHandle, RAIL_CalMask_t calEnable)
Initialize RAIL calibration.

RAIL_Status_t	RAIL_Calibrate (RAIL_Handle_t railHandle, RAIL_CalValues_t *calValues, RAIL_CalMask_t calForce) Start the calibration process.
RAIL_CalMask_t	RAIL_GetPendingCal (RAIL_Handle_t railHandle) Return the current set of pending calibrations.
RAIL_Status_t	RAIL_ApplyIrCalibration (RAIL_Handle_t railHandle, uint32_t imageRejection) Apply a given image rejection calibration value.
RAIL_Status_t	RAIL_ApplyIrCalibrationAlt (RAIL_Handle_t railHandle, RAIL_IrCalValues_t *imageRejection, RAIL_AntennaSel_t rfPath) Apply given image rejection calibration values.
RAIL_Status_t	RAIL_CalibrateIr (RAIL_Handle_t railHandle, uint32_t *imageRejection) Run the image rejection calibration.
RAIL_Status_t	RAIL_CalibrateIrAlt (RAIL_Handle_t railHandle, RAIL_IrCalValues_t *imageRejection, RAIL_AntennaSel_t rfPath) Run the image rejection calibration.
RAIL_Status_t	RAIL_CalibrateTemp (RAIL_Handle_t railHandle) Run the temperature calibration.
RAIL_Status_t	RAIL_CalibrateHFXO (RAIL_Handle_t railHandle, int8_t *crystalPPMError) Performs HFXO compensation.
void	RAIL_EnablePaCal (bool enable) Enable/disable the PA calibration.
RAIL_Status_t	RAIL_BLE_CalibrateIr (RAIL_Handle_t railHandle, uint32_t *imageRejection) Calibrate image rejection for Bluetooth Low Energy.
RAIL_Status_t	RAIL_IEEE802154_CalibrateIr2p4Ghz (RAIL_Handle_t railHandle, uint32_t *imageRejection) Calibrate image rejection for IEEE 802.15.4 2.4 GHz.
RAIL_Status_t	RAIL_IEEE802154_CalibrateIrSubGhz (RAIL_Handle_t railHandle, uint32_t *imageRejection) Calibrate image rejection for IEEE 802.15.4 915 MHz and 868 MHz.

Macros

#define	RAIL_CAL_TEMP_VCO (0x00000001U) EFR32-specific temperature calibration bit.
#define	RAIL_CAL_TEMP_HFXO (0x00000002U) EFR32-specific HFXO temperature check bit.
#define	RAIL_CAL_COMPENSATE_HFXO (0x00000004U) EFR32-specific HFXO compensation bit.
#define	RAIL_CAL_RX_IRCAL (0x00010000U) EFR32-specific IR calibration bit.
#define	RAIL_CAL_OFDM_TX_IRCAL (0x00100000U) EFR32-specific Tx IR calibration bit.
#define	RAIL_CAL_ONETIME_IRCAL (RAIL_CAL_RX_IRCAL RAIL_CAL_OFDM_TX_IRCAL) A mask to run EFR32-specific IR calibrations.
#define	RAIL_CAL_TEMP (RAIL_CAL_TEMP_VCO RAIL_CAL_TEMP_HFXO RAIL_CAL_COMPENSATE_HFXO) A mask to run temperature-dependent calibrations.

#define	RAIL_CAL_ONETIME (RAIL_CAL_ONETIME_IRCAL) A mask to run one-time calibrations.
#define	RAIL_CAL_PERF (0) A mask to run optional performance calibrations.
#define	RAIL_CAL_OFFLINE (RAIL_CAL_ONETIME_IRCAL) A mask for calibrations that require the radio to be off.
#define	RAIL_CAL_ALL (RAIL_CAL_TEMP RAIL_CAL_ONETIME) A mask to run all possible calibrations for this chip.
#define	RAIL_CAL_ALL_PENDING (0x00000000U) A mask to run all pending calibrations.
#define	RAIL_CAL_INVALID_VALUE (0xFFFFFFFFU) An invalid calibration value.
#define	RAIL_MAX_RF_PATHS 2 Indicates the maximum number of RF Paths supported across all platforms.
#define	RAIL_IRCALVALUES_RX_UNINIT undefined A define to set all RAIL_RxIrCalValues_t values to uninitialized.
#define	RAIL_IRCALVALUES_TX_UNINIT undefined A define to set all RAIL_TxIrCalValues_t values to uninitialized.
#define	RAIL_IRCALVALUES_UNINIT undefined A define to set all RAIL_IrCalValues_t values to uninitialized.
#define	RAIL_IRCALVAL (irCalStruct, rfPath) A define allowing Rx calibration value access compatibility between non-OFDM and OFDM platforms.
#define	RAIL_CALVALUES_UNINIT RAIL_IRCALVALUES_UNINIT A define to set all RAIL_CalValues_t values to uninitialized.
#define	RAIL_PACTUNE_IGNORE (255U) Use this value with either TX or RX values in RAIL_SetPaCTune() to use whatever value is already set and do no update.

Typedef Documentation

RAIL_CalMask_t

```
RAIL_CalMask_t
```

A calibration mask type.

This type is a bitmask of different RAIL calibration values. The exact meaning of these bits depends on what a particular chip supports.

Definition at line 4674 of file `common/rail_types.h`

RAIL_RxIrCalValues_t

```
typedef uint32_t RAIL_RxIrCalValues_t[2] [2]
```

RAIL_RxIrCalValues_t.

RX IR calibration values.

Platforms with fewer `RAIL_RF_PATHS` than `RAIL_MAX_RF_PATHS` will only respect and update `RAIL_RF_PATHS` indices and ignore the rest.

Definition at line 4722 of file `common/rail_types.h`

RAIL_CalValues_t

RAIL_CalValues_t

A calibration value structure.

This structure contains the set of persistent calibration values for EFR32. You can set these beforehand and apply them at startup to save the time required to compute them. Any of these values may be set to `RAIL_CAL_INVALID_VALUE` to force the code to compute that calibration value.

Definition at line 4807 of file `common/rail_types.h`

Function Documentation

RAIL_ConfigCal

RAIL_Status_t RAIL_ConfigCal (`RAIL_Handle_t` railHandle, `RAIL_CalMask_t` calEnable)

Initialize RAIL calibration.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	calEnable	A bitmask that indicates which calibrations to enable for a callback notification. The exact meaning of these bits is radio-specific.

Returns

- Status code indicating success of the function call.

Calibration initialization provides the calibration settings that correspond to the current radio configuration.

Definition at line 5190 of file `common/rail.h`

RAIL_Calibrate

RAIL_Status_t RAIL_Calibrate (`RAIL_Handle_t` railHandle, `RAIL_CalValues_t` * calValues, `RAIL_CalMask_t` calForce)

Start the calibration process.

Parameters

[in]	railHandle	A RAIL instance handle.

[inout]	calValues	A pointer to a structure of calibration values to apply. If a valid calibration structure is provided and the structure contains valid calibration values, those values will be applied to the hardware and the RAIL library will cache those values for use again later. If a valid calibration structure is provided and the structure contains a calibration value of RAIL_CAL_INVALID_VALUE for the desired calibration, the desired calibration will run, the calibration values structure will be updated with a valid calibration value, and the RAIL library will cache that value for use again later. If a NULL pointer is provided, the desired calibration will run and the RAIL library will cache that value for use again later. However, the valid calibration value will not be returned to the application.
[in]	calForce	A mask to force specific calibration(s) to execute. To run all pending calibrations, use the value RAIL_CAL_ALL_PENDING . Only the calibrations specified will run, even if not enabled during initialization.

Returns

- Status code indicating success of the function call.

If calibrations were performed previously and the application saves the calibration values (i.e., call this function with a calibration values structure containing calibration values of [RAIL_CAL_INVALID_VALUE](#) before a reset), the application can later bypass the time it would normally take to recalibrate hardware by reusing previous calibration values (i.e., call this function with a calibration values structure containing valid calibration values after a reset).

Silicon Labs recommends calling this function from the application main loop.

If multiple protocols are used, this function will make the given railHandle active, if not already, and perform calibration. If called during a protocol switch, to perform an IR calibration for the first time, it will return [RAIL_STATUS_INVALID_STATE](#), in which case the application must defer calibration until after the protocol switch is complete.

Note

- Instead of this function, consider using the individual calibration-specific functions. Using the individual functions will allow for better dead-stripping if not all calibrations are run.
- Some calibrations should only be executed when the radio is IDLE. See chip-specific documentation for more details.

Definition at line 5237 of file `common/rail.h`

RAIL_GetPendingCal

```
RAIL_CalMask_t RAIL_GetPendingCal (RAIL\_Handle\_t railHandle)
```

Return the current set of pending calibrations.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- A mask of all pending calibrations that the user has been asked to perform.

This function returns a full set of pending calibrations. The only way to clear pending calibrations is to perform them using the [RAIL_Calibrate\(\)](#) API with the appropriate list of calibrations.

Definition at line 5252 of file `common/rail.h`

RAIL_ApplyIrCalibration

```
RAIL_Status_t RAIL_ApplyIrCalibration (RAIL\_Handle\_t railHandle, uint32_t imageRejection)
```

Apply a given image rejection calibration value.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	imageRejection	The image rejection value to apply.

Returns

- Status code indicating success of the function call.

Take an image rejection calibration value and apply it. This value should be determined from a previous run of [RAIL_CalibrateIr](#) on the same physical device with the same radio configuration. The imageRejection value will also be stored to the [RAIL_ChannelConfigEntry_t::attr](#), if possible.

If multiple protocols are used, this function will return [RAIL_STATUS_INVALID_STATE](#) if it is called and the given railHandle is not active. In that case, the caller must attempt to re-call this function later.

DeprecatedPlease use [RAIL_ApplyIrCalibrationAlt](#) instead.

Definition at line 5272 of file `common/rail.h`

RAIL_ApplyIrCalibrationAlt

RAIL_Status_t RAIL_ApplyIrCalibrationAlt ([RAIL_Handle_t](#) railHandle, [RAIL_IrCalValues_t](#) * imageRejection, [RAIL_AntennaSel_t](#) rfPath)

Apply given image rejection calibration values.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	imageRejection	A pointer to the image rejection values to apply.
[in]	rfPath	RF path(s) to calibrate.

Returns

- Status code indicating success of the function call.

Take image rejection calibration values and apply them. These values should be determined from a previous run of [RAIL_CalibrateIrAlt\(\)](#) on the same physical device with the same radio configuration. The imageRejection values will also be stored to the [RAIL_ChannelConfigEntry_t::attr](#), if possible.

Note

- : To make sure the imageRejection values are stored/configured correctly, [RAIL_ConfigAntenna\(\)](#) should be called before calling this API.

If multiple protocols are used, this function will return [RAIL_STATUS_INVALID_STATE](#) if it is called and the given railHandle is not active. In that case, the caller must attempt to re-call this function later.

Definition at line 5295 of file `common/rail.h`

RAIL_CalibrateIr

RAIL_Status_t RAIL_CalibrateIr ([RAIL_Handle_t](#) railHandle, uint32_t * imageRejection)

Run the image rejection calibration.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	imageRejection	The result of the image rejection calibration.

Returns

- Status code indicating success of the function call.

Run the image rejection calibration and apply the resulting value. If the imageRejection parameter is not NULL, store the value at that location. The imageRejection value will also be stored to the [RAIL_ChannelConfigEntry_t::attr](#), if possible. This is a long-running calibration that adds significant code space when run and can be run with a separate firmware image on each device to save code space in the final image. Silicon Labs recommends calling this function from the application main loop.

If multiple protocols are used, this function will make the given railHandle active, if not already, and perform calibration. If called during a protocol switch, it will return [RAIL_STATUS_INVALID_STATE](#). In this case, [RAIL_ApplyIrCalibration](#) may be called to apply a previously determined IR calibration value, or the app must defer calibration until the protocol switch is complete.

DeprecatedPlease use [RAIL_CalibrateIrAlt](#) instead.

Definition at line 5324 of file `common/rail.h`

RAIL_CalibrateIrAlt

```
RAIL_Status_t RAIL_CalibrateIrAlt (RAIL_Handle_t railHandle, RAIL_IrCalValues_t * imageRejection, RAIL_AntennaSel_t rfPath)
```

Run the image rejection calibration.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	imageRejection	A pointer to the image rejection results.
[in]	rfPath	RF path(s) to calibrate.

Returns

- Status code indicating success of the function call.

Run the image rejection calibration and apply the resulting values. If the imageRejection parameter is not NULL, store the values at that location. The imageRejection values will also be stored to the [RAIL_ChannelConfigEntry_t::attr](#), if possible. This is a long-running calibration that adds significant code space when run and can be run with a separate firmware image on each device to save code space in the final image. Silicon Labs recommends calling this function from the application main loop.

Note

- : To make sure the imageRejection values are stored/configured correctly, [RAIL_ConfigAntenna\(\)](#) should be called before calling this API.

If multiple protocols are used, this function will return [RAIL_STATUS_INVALID_STATE](#) if it is called and the given railHandle is not active. In that case, the caller must attempt to re-call this function later.

Definition at line 5351 of file `common/rail.h`

RAIL_CalibrateTemp

```
RAIL_Status_t RAIL_CalibrateTemp (RAIL_Handle_t railHandle)
```

Run the temperature calibration.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

Run the temperature calibration, which needs to recalibrate the synth if the temperature crosses 0C or the temperature delta since the last calibration exceeds 70C while in receive. RAIL will run the VCO calibration automatically upon entering receive or transmit states, so the application can omit this calibration if the stack re-enters receive or transmit with enough frequency to avoid reaching the temperature delta. If the application does not calibrate for temperature, it's possible to miss receive packets due to a drift in the carrier frequency.

If multiple protocols are used, this function will return [RAIL_STATUS_INVALID_STATE](#) if it is called and the given railHandle is not active. In that case, the calibration will be automatically performed next time the radio enters receive.

Note

- If RX channel hopping is enabled this function may trigger a channel hop, which can result in [RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE](#) occurring before it returns.

Definition at line 5379 of file common/rail.h

RAIL_CalibrateHFXO

RAIL_Status_t RAIL_CalibrateHFXO ([RAIL_Handle_t](#) railHandle, int8_t * crystalPPMError)

Performs HFXO compensation.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	crystalPPMError	A pointer for RAIL to store the current deviation that has been corrected, measured in PPM. May be NULL.

Returns

- Status code indicating the result of the function call.

Compute the PPM correction using the thermistor value available when [RAIL_EVENT_THERMISTOR_DONE](#) occurs, after [RAIL_StartThermistorMeasurement\(\)](#) call. Then correct the RF frequency as well as TX and RX sampling.

This function calls the following RAIL functions in sequence saving having to call them individually:

- [RAIL_ConvertThermistorImpedance\(\)](#)
- [RAIL_ComputeHFXOPPMError\(\)](#)
- [RAIL_CompensateHFXO\(\)](#)

Note

- This function makes the radio idle.

Definition at line 5402 of file common/rail.h

RAIL_EnablePaCal

void RAIL_EnablePaCal (bool enable)

Enable/disable the PA calibration.

Parameters

[in]	enable	Enables/disables the PA calibration.

Enabling will ensure that the PA power remains constant chip-to-chip. This feature is enabled by default on Series-2 platforms.

On EFR32xG21 and EFR32xG24 platforms, particularly the 20 dBm variants, it is recommended to keep this feature enabled in order to utilize the chip-specific calibrations designated for the chip.

The chip-specific define SL_RAIL_UTIL_PA_CALIBRATION_ENABLE in the [Power Amplifier \(PA\) Utility](#) plugin also enables/disables PA calibrations on initialization, which can override the default state of the feature.

Note

- Call this function before [RAIL_ConfigTxPower\(\)](#) if this feature is not desired.

Definition at line 5423 of file common/rail.h

RAIL_BLE_CalibrateIr

```
RAIL_Status_t RAIL_BLE_CalibrateIr (RAIL\_Handle\_t railHandle, uint32_t * imageRejection)
```

Calibrate image rejection for Bluetooth Low Energy.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	imageRejection	A pointer to where the result of the image rejection calibration will be stored.

Returns

- Status code indicating success of the function call.

Some chips have protocol-specific image rejection calibrations programmed into their flash. This function will either get the value from flash and apply it, or run the image rejection algorithm to find the value.

Definition at line 1854 of file protocol/ble/rail_ble.h

RAIL_IEEE802154_CalibrateIr2p4Ghz

```
RAIL_Status_t RAIL_IEEE802154_CalibrateIr2p4Ghz (RAIL\_Handle\_t railHandle, uint32_t * imageRejection)
```

Calibrate image rejection for IEEE 802.15.4 2.4 GHz.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	imageRejection	A pointer to the result of the image rejection calibration.

Returns

- Status code indicating success of the function call.

Some chips have protocol-specific image rejection calibrations programmed into their flash. This function will either get the value from flash and apply it, or run the image rejection algorithm to find the value.

Definition at line 1807 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_CalibrateIrSubGhz

RAIL_Status_t RAIL_IEEE802154_CalibrateIrSubGhz ([RAIL_Handle_t](#) railHandle, uint32_t * imageRejection)

Calibrate image rejection for IEEE 802.15.4 915 MHz and 868 MHz.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	imageRejection	The result of the image rejection calibration.

Returns

- Status code indicating success of the function call.

Some chips have protocol-specific image rejection calibrations programmed into their flash. This function will either get the value from flash and apply it, or run the image rejection algorithm to find the value.

DeprecatedPlease use [RAIL_CalibrateIrAlt](#) instead.

Definition at line 1825 of file protocol/ieee802154/rail_ieee802154.h

RAIL_TxIrCalValues_t

A Tx IR calibration value structure.

This definition contains the set of persistent calibration values for OFDM on EFR32. You can set these beforehand and apply them at startup to save the time required to compute them. Any of these values may be set to [RAIL_CAL_INVALID_VALUE](#) to force the code to compute that calibration value.

Only supported on platforms with [RAIL_SUPPORTS_OFDM_PA](#) enabled.

Public Attributes

uint32_t	dcOffsetIQ
	TXIRCAL result.
uint32_t	phiEpsilon
	TXIRCAL result.

Public Attribute Documentation

dcOffsetIQ

uint32_t RAIL_TxIrCalValues_t::dcOffsetIQ

TXIRCAL result.

Definition at line 4748 of file `common/rail_types.h`

phiEpsilon

uint32_t RAIL_TxIrCalValues_t::phiEpsilon

TXIRCAL result.

Definition at line 4750 of file `common/rail_types.h`

RAIL_IrCalValues_t

An IR calibration value structure.

This definition contains the set of persistent calibration values for EFR32. You can set these beforehand and apply them at startup to save the time required to compute them. Any of these values may be set to [RAIL_CAL_INVALID_VALUE](#) to force the code to compute that calibration value.

Public Attributes

RAIL_RxIrCalValue s_t	rxIrCalValues RX Image Rejection (IR) calibration value(s)
RAIL_TxIrCalValue s_t	txIrCalValues TX Image Rejection (IR) calibration value(s) for OFDM.

Public Attribute Documentation

rxIrCalValues

RAIL_RxIrCalValues_t RAIL_IrCalValues_t::rxIrCalValues

RX Image Rejection (IR) calibration value(s)

Definition at line 4775 of file common/rail_types.h

txIrCalValues

RAIL_TxIrCalValues_t RAIL_IrCalValues_t::txIrCalValues

TX Image Rejection (IR) calibration value(s) for OFDM.

Definition at line 4777 of file common/rail_types.h

EFR32xG2x

EFR32xG2x

EFR32xG2x-specific Calibrations.

Modules

[RAIL_ChannelConfigEntryAttr](#)

Macros

#define	RAIL_RF_PATHS_2P4GIG 0	Indicates the number of 2.4 GHz RF Paths supported.
#define	RAIL_RF_PATHS_SUBGIG 0	Indicates the number of Sub-GHz RF Paths supported.
#define	RAIL_RF_PATHS (RAIL_RF_PATHS_SUBGIG + RAIL_RF_PATHS_2P4GIG)	Indicates the number of RF Paths supported.
#define	RADIO_CONFIG_ENABLE_IRCAL_MULTIPLE_RF_PATHS 0	Indicates this version of RAIL supports IR calibration on multiple RF paths Needed for backwards compatibility.

RAIL_ChannelConfigEntryAttr

A channel configuration entry attribute structure.

Items listed are designed to be altered and updated during run-time.

Public Attributes

RAIL_RxIrCalValue

s_t

calValues

IR calibration attributes specific to each channel configuration entry.

Public Attribute Documentation

calValues

RAIL_RxIrCalValues_t RAIL_ChannelConfigEntryAttr_t::calValues

IR calibration attributes specific to each channel configuration entry.

RAIL_SUPPORTS_OFDM_PA

Definition at line 436 of file chip/efr32/efr32xg2x/rail_chip_specific.h

Slxx3xx

Slxx3xx

Slxx3xx-specific Calibrations.

Modules

[RAIL_ChannelConfigEntryAttr](#)

Macros

#define	RAIL_RF_PATHS_2P4GIG 0	Indicates the number of 2.4 GHz RF Paths supported.
#define	RAIL_RF_PATHS_SUBGIG 0	Indicates the number of Sub-GHz RF Paths supported.
#define	RAIL_RF_PATHS (RAIL_RF_PATHS_SUBGIG + RAIL_RF_PATHS_2P4GIG)	Indicates the number of RF Paths supported.
#define	RADIO_CONFIG_ENABLE_IRCAL_MULTIPLE_RF_PATHS 0	Indicates this version of RAIL supports IR calibration on multiple RF paths Needed for backwards compatibility.

Chip-Specific

Chip-Specific

Chip-Specific RAIL APIs, types, and information.

Modules

[RAIL_EFR32xG2x_Interrupts](#)

RAIL_EFR32xG2x_Interrupts

EFR32xG2x-specific interrupt sources.

Below are the interrupt handlers implemented within the RAIL library for EFR32 Series 2. See [EFR32](#) for more details about interrupt initialization and priorities. For more information about the interrupt vectors and where they are on your specific chip, see the `startup_efr32xg2xx.c` file for your chip. These can be found under the Device/SiliconLabs folder in your release.

Warnings

- [RAC_RSM_IRQHandler\(\)](#) and [RAC_SEQ_IRQHandler\(\)](#) must be set to the same priority level.

Public Functions

void	FRC_PRI_IRQHandler(void)	RAIL interrupt handler for the high-priority FRC_PRI interrupt source.
void	FRC_IRQHandler(void)	RAIL interrupt handler for the normal-priority FRC interrupt source.
void	MODEM_IRQHandler(void)	RAIL interrupt handler for the MODEM interrupt source.
void	RAC_SEQ_IRQHandler(void)	RAIL interrupt handler for the SEQ interrupt source.
void	RAC_RSM_IRQHandler(void)	RAIL interrupt handler for the RAC_RSM interrupt source.
void	BUFC_IRQHandler(void)	RAIL interrupt handler for the BUFC interrupt source.
void	AGC_IRQHandler(void)	RAIL interrupt handler for the AGC interrupt source.
void	PROTIMER_IRQHandler(void)	RAIL interrupt handler for the PROTIMER interrupt source.
void	SYNTH_IRQHandler(void)	RAIL interrupt handler for the SYNTH interrupt source.
void	RFSENSE_IRQHandler(void)	RAIL interrupt handler for the RFSENSE interrupt source.
void	PRORTC_IRQHandler(void)	RAIL interrupt handler for the PRORTC interrupt source.
void	HOSTMAILBOX_IRQHandler(void)	RAIL interrupt handler for the HOSTMAILBOX interrupt source.
void	RDMAILBOX_IRQHandler(void)	RAIL interrupt handler for the RDMAILBOX interrupt source.
void	RFECA0_IRQHandler(void)	RAIL interrupt handler for the RFECA0 interrupt source.
void	RFECA1_IRQHandler(void)	RAIL interrupt handler for the RFECA1 interrupt source.

- void

[RFTIMER_IRQHandler](#)(void)
RAIL interrupt handler for the RFTIMER interrupt source.
- void

[SOFTM_IRQHandler](#)(void)
RAIL interrupt handler for the SOFTM interrupt source.
- void

[RFLDMA_IRQHandler](#)(void)
RAIL interrupt handler for the RFDMA interrupt source.
- void

[SYSRTC_SEQ_IRQHandler](#)(void)
RAIL interrupt handler for the SYSRTC SEQ interrupt source.
- void

[EMUDG_IRQHandler](#)(void)
RAIL interrupt handler for the EMUDG interrupt source.

Public Function Documentation

FRC_PRI_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::FRC_PRI_IRQHandler (void )
```

RAIL interrupt handler for the high-priority FRC_PRI interrupt source.

Parameters

N/A		

This source handles interrupts from the FRC High Priority. It is not used at this time.

Note

- The interrupt vector should still point to the RAIL interrupt handler for compatibility with future RAIL releases.

Definition at line 53 of file `chip/efr32/efr32xg2x/railLefr32xg2x_interrupts.h`

FRC_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::FRC_IRQHandler (void )
```

RAIL interrupt handler for the normal-priority FRC interrupt source.

Parameters

N/A		

This source handles packet completion events for the transmit side. The receive side events are queued and processed in the [BUFC_IRQHandler](#). The following events are currently in use:

- TX Completed
 - This event cannot be disabled and is used to clean up after a transmit event. If enabled, one of the following events will be generated after transmit completion: [RAIL_EVENT_TX_PACKET_SENT](#), [RAIL_EVENT_TX_ABORTED](#), [RAIL_EVENT_TX_BLOCKED](#), or [RAIL_EVENT_TX_UNDERFLOW](#).
- TX Ack Completed
 - This event is only turned on if the user calls [RAIL_ConfigEvents\(\)](#) with one of [RAIL_EVENT_TXACK_PACKET_SENT](#), [RAIL_EVENT_TXACK_ABORTED](#), [RAIL_EVENT_TXACK_BLOCKED](#), or [RAIL_EVENT_TXACK_UNDERFLOW](#). If enabled, the corresponding event will be generated when an Ack completes.

Definition at line 75 of file `chip/efr32/efr32xg2x/railLefr32xg2x_interrupts.h`

MODEM_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::MODEM_IRQHandler (void )
```

RAIL interrupt handler for the MODEM interrupt source.

Parameters

N/A		

This source handles demodulator and modulator status and error events. The following events are currently in use:

- Preamble Detection
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_RX_PREAMBLE_DETECT](#). This event will call the `RAIL_eventsCallback` when a preamble is detected by the demodulator if enabled.
- Preamble Lost
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_RX_PREAMBLE_LOST](#). This event will call the `RAIL_eventsCallback` when a preamble is lost by the demodulator if enabled.
- Timing Detection
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_RX_TIMING_DETECT](#). This event will call the `RAIL_eventsCallback` when timing is detected by the demodulator if enabled.
- Timing Lost
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_RX_TIMING_LOST](#). This event will call the `RAIL_eventsCallback` when timing is lost by the demodulator if enabled.
- Sync 1 Detection
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_RX_SYNC1_DETECT](#). This event will call the `RAIL_eventsCallback` when sync word 1 is detected by the demodulator if enabled.
- Sync 2 Detection
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_RX_SYNC2_DETECT](#). This event will call the `RAIL_eventsCallback` when sync word 2 is detected by the demodulator if enabled.

Definition at line 112 of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

RAC_SEQ_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::RAC_SEQ_IRQHandler (void )
```

RAIL interrupt handler for the SEQ interrupt source.

Parameters

N/A		

This source handles interrupts generated from the radio sequencer. The following events are currently in use:

- RX timeout
 - This event cannot be disabled and occurs when a receive timeout, as configured via [RAIL_SetStateTiming\(\)](#)[RAIL_StateTiming_t::rxSearchTimeout](#) or [RAIL_StateTiming_t::txToRxSearchTimeout](#), expires and the radio exits receive. This event will call the generic callback if [RAIL_EVENT_RX_TIMEOUT](#) is enabled.
- Ack Timeout
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_RX_ACK_TIMEOUT](#). This event is only available when using [Auto-Ack](#) or [IEEE 802.15.4](#) and will call the `eventsCallback` whenever an Ack packet is not received within the Ack timeout window.
- Address Filtering Passed
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_RX_FILTER_PASSED](#). This event is only enabled when using [Address Filtering](#) and will call the generic callback whenever a packet passes the filtering criteria.
- IEEE802154 Frame Pending
 -

This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#). This event is only available when in [IEEE 802.15.4](#) mode and will call the `eventsCallback` whenever a data request packet is received so you can set a frame-pending indicator in the outgoing Ack.

- Temperature Calibration
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_CAL_NEEDED](#). This event will fire whenever the temperature changes enough to require recalibration during the RX state.

Definition at line 152 of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

RAC_RSM_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::RAC_RSM_IRQHandler (void )
```

RAIL interrupt handler for the RAC_RSM interrupt source.

Parameters

N/A		

This source handles interrupts generated from the radio state machine. The following events are currently in use:

- Radio State Changed
 - Enabled in some debug modes.

Definition at line 163 of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

BUFC_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::BUFC_IRQHandler (void )
```

RAIL interrupt handler for the BUFC interrupt source.

Parameters

N/A		

This source handles interrupts generated from the radio buffer controller. The following events are currently in use:

- RX Buffer Threshold
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_RX_FIFO_ALMOST_FULL](#). This event will trigger an `eventsCallback` when the contents of the receive buffer exceed the receive buffer threshold set by [RAIL_SetRxFifoThreshold\(\)](#).
- RX Buffer Underflow
 - This event cannot be disabled and indicates that too much data was read from the RX buffer. It should not occur in normal operation as all receive APIs protect against it.
- RX Buffer Corruption
 - This event cannot be disabled and indicates the pointers in the RX buffer have become corrupt. This event should not occur in normal operation.
- TX Buffer Threshold
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_TX_FIFO_ALMOST_EMPTY](#). This event will trigger an `eventsCallback` when the contents of the transmit buffer fall beneath the transmit threshold set by [RAIL_SetTxFifoThreshold\(\)](#).
- TX Buffer Overflow
 - This event cannot be disabled and indicates too much data was placed into the TX buffer. This event should not occur in normal operation as all transmit APIs protect against it.
- TX Buffer Underflow
 - This event cannot be disabled and occurs when the transmit FIFO runs out of bytes before the end of the packet. This event will always perform any necessary cleanup and will call the `eventsCallback` if [RAIL_EVENT_TX_UNDERFLOW](#) is

enabled.

- TX Buffer Corruption
 - This event cannot be disabled and indicates the pointers in the TX buffer have become corrupt. This event should not occur in normal operation.
- RX Entry Buffer Threshold
 - This event cannot be disabled when in receive mode and is used to indicate that a new packet is available in the receive buffer. This event will call the eventsCallback with any of the following if they are enabled: [RAIL_EVENT_RX_PACKET_RECEIVED](#), [RAIL_EVENT_RX_ADDRESS_FILTERED](#), [RAIL_EVENT_RX_PACKET_ABORTED](#), [RAIL_EVENT_RX_FRAME_ERROR](#), or [RAIL_EVENT_RX_FIFO_OVERFLOW](#). The [RAIL_EVENT_RX_ADDRESS_FILTERED](#) event is only used with [Address Filtering](#) to indicate termination of a packet that does not meet the filtering criteria.
- RX Entry Buffer Underflow
 - This event cannot be disabled and indicates that too much data was read from the RX entry buffer. This event should not occur in normal operation.
- RX Entry Buffer Overflow
 - This event cannot be disabled and will fire one time if the RX entry buffer overflows. It will cause the [RAIL_EVENT_RX_FIFO_OVERFLOW](#) event to happen later after all valid entries are processed in the RX entry buffer.
- RX Entry Buffer Underflow
 - This event cannot be disabled and indicates that too much data was read from the RX entry buffer. This event should not occur in normal operation.
- RX Entry Buffer Corruption
 - This event cannot be disabled and indicates the pointers in the RX entry buffer have become corrupt. This event should not occur in normal operation.

Definition at line 242 of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

AGC_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::AGC_IRQHandler (void )
```

RAIL interrupt handler for the AGC interrupt source.

Parameters

N/A		

This source handles interrupts generated from the automatic gain control block. The following events are currently in use:

- RSSI Valid
 - This event is enabled by [RAIL_StartAverageRssi\(\)](#) and occurs when the first RSSI sample is ready after going into RX state. It will cause the eventsCallback to be called with [RAIL_EVENT_RSSI_AVERAGE_DONE](#) set.

Definition at line 256 of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

PROTIMER_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::PROTIMER_IRQHandler (void )
```

RAIL interrupt handler for the PROTIMER interrupt source.

Parameters

N/A		

This source handles interrupts generated from the RAIL timebase (PROTIMER). The following events are currently in use:

- Scheduled TX/RX
 - This event cannot be disabled and occurs when a scheduled transmit or receive begins.
- RX scheduled window end timeout

This event cannot be disabled and occurs when the scheduled receive window ends and the radio exits receive. This event will call the eventsCallback if [RAIL_EVENT_RX_SCHEDULED_RX_END](#) is enabled.

- Transmit with CCA Failed
 - This event cannot be disabled and occurs when CCA indicates the channel is busy. This event will call the eventsCallback if [RAIL_EVENT_TX_CHANNEL_BUSY](#) is enabled.
- Transmit with CCA Success
 - This event cannot be disabled and occurs when CCA indicates the channel is clear and the packet will be transmitted. This event will call the eventsCallback if [RAIL_EVENT_TX_CHANNEL_CLEAR](#) is enabled.
- Starting a CCA check
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_TX_START_CCA](#). It will occur when the radio begins warming up for a CCA check.
- CCA Retry
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_TX_CCA_RETRY](#). It will occur whenever CSMA or LBT have a failed channel check and are going to begin retrying.
- RAIL Timer
 - This event is enabled by [RAIL_SetTimer\(\)](#) and will call the provided callback when the timer expires.

Definition at line 298 of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

SYNTH_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::SYNTH_IRQHandler (void )
```

RAIL interrupt handler for the SYNTH interrupt source.

Parameters

N/A		

This source handles interrupts generated from the synth module. The following events are currently in use:

- Radio Sequencer Handshake
 - This event is used in to keep the main core awake during some radio operations.

Definition at line 310 of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

RFSENSE_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::RFSENSE_IRQHandler (void )
```

RAIL interrupt handler for the RFSENSE interrupt source.

Parameters

N/A		

This source handles interrupts generated from the RF Sense module. This source is not available on all EFR32 Series 2 devices. The following events are currently in use:

- RF Sensed
 - This event is enabled by when the RFSENSE block is initialized using [RAIL_StartRfSense\(\)](#) and can be disabled by turning off the RFSENSE block using [RAIL_StartRfSense\(\)](#) with the band set to [RAIL_RFSENSE_OFF](#). This event will possibly generate a callback provided to [RAIL_StartRfSense\(\)](#) whenever the RFSENSE block detects RF energy.

Definition at line 326 of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

PRORTC_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::PRORTC_IRQHandler (void )
```

RAIL interrupt handler for the PRORTC interrupt source.

Parameters

N/A		

This source handles interrupts from the internal RTC source on chips that support it (EFR32xG21 and EFR32xG22). It is not used at this time.

Note

- The interrupt vector should still point to the RAIL interrupt handler for compatibility with future RAIL releases.

Definition at line 337 of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

HOSTMAILBOX_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::HOSTMAILBOX_IRQHandler (void )
```

RAIL interrupt handler for the HOSTMAILBOX interrupt source.

Parameters

N/A		

This source handles interrupts generated from the mailbox between the radio sequencer and main core. This source is not available on all EFR32 Series 2 devices. The following events are currently in use:

- Mailbox Event
 - Mailbox message from the radio sequencer to the main core.

Definition at line 350 of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

RDMAILBOX_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::RDMAILBOX_IRQHandler (void )
```

RAIL interrupt handler for the RDMAILBOX interrupt source.

Parameters

N/A		

This interrupt source was renamed. See [HOSTMAILBOX_IRQHandler](#) for more details.

Definition at line 358 of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

RFECA0_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::RFECA0_IRQHandler (void )
```

RAIL interrupt handler for the RFECA0 interrupt source.

Parameters

N/A		

Currently, there are no interrupt sources enabled within this group. This source is not available on all EFR32 Series 2 devices.

Note

- The interrupt vector should still point to the RAIL interrupt handler for compatibility with future RAIL releases.

Definition at line 369 of file chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h

RFECA1_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::RFECA1_IRQHandler (void )
```

RAIL interrupt handler for the RFECA1 interrupt source.

Parameters

N/A		

Currently, there are no interrupt sources enabled within this group. This source is not available on all EFR32 Series 2 devices.

Note

- The interrupt vector should still point to the RAIL interrupt handler for compatibility with future RAIL releases.

Definition at line 380 of file chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h

RFTIMER_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::RFTIMER_IRQHandler (void )
```

RAIL interrupt handler for the RFTIMER interrupt source.

Parameters

N/A		

Currently, there are no interrupt sources enabled within this group. This source is not available on all EFR32 Series 2 devices.

Note

- The interrupt vector should still point to the RAIL interrupt handler for compatibility with future RAIL releases.

Definition at line 391 of file chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h

SOFTM_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::SOFTM_IRQHandler (void )
```

RAIL interrupt handler for the SOFTM interrupt source.

Parameters

N/A		

This source handles demodulator status and error events on some phys configurations. This source is not available on all EFR32 Series 2 devices.

The following events are currently in use:

- Preamble Detection
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_RX_PREAMBLE_DETECT](#). This event will call the RAIL eventsCallback when a preamble is detected by the demodulator if enabled.
- Preamble Lost
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_RX_PREAMBLE_LOST](#). This event will call the RAIL eventsCallback when a preamble is lost by the demodulator if enabled.
- Timing Detection
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_RX_TIMING_DETECT](#). This event will call the RAIL eventsCallback when timing is detected by the demodulator if enabled.
- Timing Lost
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_RX_TIMING_LOST](#). This event will call the RAIL eventsCallback when timing is lost by the demodulator if enabled.
- Sync 1 Detection
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_RX_SYNC1_DETECT](#). This event will call the RAIL eventsCallback when sync word 1 is detected by the demodulator if enabled.
- Sync 2 Detection
 - This event is enabled by [RAIL_ConfigEvents\(\)](#) when given [RAIL_EVENT_RX_SYNC2_DETECT](#). This event will call the RAIL eventsCallback when sync word 2 is detected by the demodulator if enabled.

Definition at line 431 of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

RFLDMA_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::RFLDMA_IRQHandler (void )
```

RAIL interrupt handler for the RFDMA interrupt source.

Parameters

N/A		

Currently, there are no interrupt sources enabled within this group. This source is not available on all EFR32 Series 2 devices.

Note

- The interrupt vector should still point to the RAIL interrupt handler for compatibility with future RAIL releases.

Definition at line 442 of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

SYSRTC_SEQ_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::SYSRTC_SEQ_IRQHandler (void )
```

RAIL interrupt handler for the SYSRTC SEQ interrupt source.

Parameters

N/A		

This source handles system rtc interrupts for the radio subsystem. This source is not available on all EFR32 Series 2 devices.

The following events are currently in use:

- SYSRTC compare channel
 - This event is enabled by the rail_power_manager and is only used during rail timebase synchronization after sleep.

Definition at line 455 of file chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h

EMUDG_IRQHandler

```
void RAIL_EFR32xG2x_Interrupts::EMUDG_IRQHandler (void )
```

RAIL interrupt handler for the EMUDG interrupt source.

Parameters

N/A		

This source handles multiple features based on temperature such as:

- HFXO compensation defined by [RAIL_SUPPORTS_HFXO_COMPENSATION](#).
 - This feature is enabled by [RAIL_ConfigHFXOCompensation\(\)](#).
- Chip thermal protection defined by [RAIL_SUPPORTS_THERMAL_PROTECTION](#).
 - This feature is enabled by [RAIL_ConfigThermalProtection\(\)](#).
- OFDM PA compensation over temperature defined by [RAIL_SUPPORTS_OFDM_PA](#)
 - This feature is automatically enabled when an OFDM PHY is loaded.
- VCO calibration
 - This feature is automatically enabled.

This source is not available on all EFR32 Series 2 devices.

Definition at line 475 of file chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h

Data Management

Data Management

Data management functions.

These functions allow the application to choose how data is presented to the application. RAIL provides data in a packet-based method or in a FIFO-based method. As originally conceived, [RAIL_DataMethod_t::PACKET_MODE](#) was designed for handling packets that fit within RAIL's FIFOs while [RAIL_DataMethod_t::FIFO_MODE](#) was designed for handling packets larger than RAIL's FIFOs could hold. Conceptually it is still useful to think of these modes this way, but functionally their distinction has become blurred by improvements in RAIL's flexibility – applications now have much more control over both receive and transmit FIFO sizes, and the FIFO-management and threshold APIs and related events are no longer restricted to [RAIL_DataMethod_t::FIFO_MODE](#) operation but can be used in [RAIL_DataMethod_t::PACKET_MODE](#) too.

The application can configure RAIL data management through [RAIL_ConfigData\(\)](#). This function allows the application to specify the type of radio data ([RAIL_TxDataSource_t](#) and [RAIL_RxDataSource_t](#)) and the method of interacting with data ([RAIL_DataMethod_t](#)). By default, RAIL configures TX and RX both with packet data source and [RAIL_DataMethod_t::PACKET_MODE](#).

For transmit, [RAIL_DataMethod_t::PACKET_MODE](#) and [RAIL_DataMethod_t::FIFO_MODE](#) are functionally the same:

- When not actively transmitting, load a packet's initial transmit data using [RAIL_WriteTxFifo\(\)](#) with reset set to true. Alternatively this data copying can be avoided by changing the transmit FIFO to an already-loaded section of memory with [RAIL_SetTxFifo\(\)](#).
- When actively transmitting, load remaining transmit data with [RAIL_WriteTxFifo\(\)](#) with reset set to false.
- If transmit packets exceed the FIFO size, set the transmit FIFO threshold through [RAIL_SetTxFifoThreshold\(\)](#). The [RAIL_Config_t::eventsCallback](#) with [RAIL_EVENT_TX_FIFO_ALMOST_EMPTY](#) will occur telling the application to load more TX packet data, if needed, to prevent a [RAIL_EVENT_TX_UNDERFLOW](#) event from occurring. One can get how much space is available in the transmit FIFO for more transmit data through [RAIL_GetTxFifoSpaceAvailable\(\)](#).
- After transmit completes, the transmit FIFO can be manually reset with [RAIL_ResetFifo\(\)](#), but this should rarely be necessary.

The transmit FIFO is specified by the application and its size is the value returned from the most recent call to [RAIL_SetTxFifo\(\)](#). The transmit FIFO is edge-based in that it only provides the [RAIL_EVENT_TX_FIFO_ALMOST_EMPTY](#) event once when the threshold is crossed in the emptying direction.

For receive, the distinction between [RAIL_DataMethod_t::PACKET_MODE](#) and [RAIL_DataMethod_t::FIFO_MODE](#) basically boils down to how unsuccessfully-received packets are handled. In [RAIL_DataMethod_t::PACKET_MODE](#), data from such packets is automatically rolled back as if the packet was never received, while in [RAIL_DataMethod_t::FIFO_MODE](#), rollback does not occur putting more onus on the application to deal with that data.

In receive [RAIL_DataMethod_t::PACKET_MODE](#) data management:

- Packet lengths are determined from the Radio Configurator configuration and can be read out at the end using [RAIL_GetRxPacketInfo\(\)](#).
- Received packet data is made available on successful packet completion via [RAIL_Config_t::eventsCallback](#) with [RAIL_EVENT_RX_PACKET_RECEIVED](#) which can then use [RAIL_GetRxPacketInfo\(\)](#) and [RAIL_GetRxPacketDetailsAlt\(\)](#) to access packet information and [RAIL_PeekRxPacket\(\)](#) to access packet data.
- FILTERED, ABORTED, or FRAMEERROR received packet data is automatically rolled back (dropped) without the application needing to worry about consuming it. The application can choose to not even be bothered with the events related to such packets: [RAIL_EVENT_RX_ADDRESS_FILTERED](#), [RAIL_EVENT_RX_PACKET_ABORTED](#), or [RAIL_EVENT_RX_FRAME_ERROR](#).

In receive [RAIL_DataMethod_t::FIFO_MODE](#) data management:

- Packet Lengths are determined from the Radio Configurator configuration or by application knowledge of packet payload structure.

- Received data can be retrieved prior to packet completion through [RAIL_ReadRxFifo\(\)](#) and is never rolled back on FILTERED, ABORTED, or FRAMEERROR packets. The application should enable and handle these events so it can flush any packet data it's already retrieved.
- After packet completion, remaining packet data for FILTERED, ABORTED, or FRAMEERROR packets remains in the FIFO and the appropriate event is triggered to the user. This data may be consumed in the callback unlike in packet mode where it is automatically rolled back. At the end of the callback all remaining data in the FIFO will be cleaned up as usual. Keep in mind that [RAIL_GetRxPacketDetailsAlt\(\)](#) provides full packet detailed information only for successfully received packets.

Common receive data management features:

- Set the receive FIFO threshold through [RAIL_SetRxFifoThreshold\(\)](#). The [RAIL_Config_t::eventsCallback](#) with [RAIL_EVENT_RX_FIFO_ALMOST_FULL](#) will occur telling the application to consume some RX packet data to prevent a [RAIL_EVENT_RX_FIFO_OVERFLOW](#) event from occurring.
- Get receive FIFO count information through [RAIL_GetRxPacketInfo\(RAIL_RX_PACKET_HANDLE_NEWEST\)](#) (or [RAIL_GetRxFifoBytesAvailable\(\)](#)).
- After receive completes and all its data has been consumed, the receive FIFO can be manually reset with [RAIL_ResetFifo\(\)](#), though this should rarely be necessary and should only be done with the radio idle.

When trying to determine an appropriate threshold, the application needs to know the size of each FIFO. The default receive FIFO is internal to RAIL with a size of 512 bytes. This can be changed, however, using [RAIL_SetRxFifo\(\)](#) and the default may be removed entirely by calling this from the [RAILCb_SetupRxFifo\(\)](#) callback. The receive FIFO event is level-based in that the [RAIL_EVENT_RX_FIFO_ALMOST_FULL](#) event will constantly pend if the threshold is exceeded. This normally means that inside this event's callback, the application should empty enough of the FIFO to go under the threshold. To defer reading the FIFO to main context, the application can disable or re-enable the receive FIFO threshold event using [RAIL_ConfigEvents\(\)](#) with the mask [RAIL_EVENT_RX_FIFO_ALMOST_FULL](#).

The receive FIFO can store multiple packets and processing of a packet can be deferred from the RAIL event callback to main-loop processing by using [RAIL_HoldRxPacket\(\)](#) in the event callback and [RAIL_ReleaseRxPacket\(\)](#) in the main-loop. On some platforms, the receive FIFO is supplemented by an internal fixed-size packet metadata FIFO that limits the number of packets RAIL and applications can hold onto for deferred processing. See chip-specific documentation, such as [EFR32](#), for more information.

Note

- When using multiprotocol the receive FIFO is reset prior to a protocol switch so held packets will be lost if not processed before then.

While [RAIL_EVENT_RX_FIFO_ALMOST_FULL](#) occurs solely based on the state of the receive FIFO used for packet data, both [RAIL_EVENT_RX_FIFO_FULL](#) and [RAIL_EVENT_RX_FIFO_OVERFLOW](#) can occur coincident with packet completion when either that or the internal packet metadata FIFO fills or overflows. [RAIL_EVENT_RX_FIFO_FULL](#) informs the application it should immediately process and free up the oldest packets/data to make room for new packets/data, reducing the possibility of packet/data loss and [RAIL_EVENT_RX_FIFO_OVERFLOW](#).

Before a packet is fully received you can always use [RAIL_PeekRxPacket\(\)](#) to look at the contents. In FIFO mode, you may also consume its data with [RAIL_ReadRxFifo\(\)](#). Remember that none of these APIs will read across a packet boundary (even in FIFO mode) so you will need to handle each received packet individually.

While RAIL defaults to [RAIL_DataMethod_t::PACKET_MODE](#), the application can explicitly initialize RAIL for [RAIL_DataMethod_t::PACKET_MODE](#) in the following manner:

```
extern RAIL_Handle_t railHandle;

void configRailForPacketModeOperation(void)
{
    RAIL_DataConfig_t railDataConfig = {
        .txSource = TX_PACKET_DATA,
        .rxSource = RX_PACKET_DATA,
        .txMethod = PACKET_MODE,
        .rxMethod = PACKET_MODE,
    };

    (void) RAIL_ConfigData(railHandle, &railDataConfig);

    // Events that can occur in Packet Mode:
    // RAIL_EVENT_TX_PACKET_SENT
    // RAIL_EVENT_RX_PACKET_RECEIVED
    // and optionally (packet data automatically dropped):
    // RAIL_EVENT_RX_ADDRESS_FILTERED
    // RAIL_EVENT_RX_PACKET_ABORTED
    // RAIL_EVENT_RX_FRAME_ERROR
    // RAIL_EVENT_RX_FIFO_OVERFLOW
    // and if enabled:
    // RAIL_EVENT_TX_UNDERFLOW
    // RAIL_EVENT_TXACK_UNDERFLOW
    // RAIL_EVENT_TX_FIFO_ALMOST_EMPTY
    // RAIL_EVENT_RX_FIFO_ALMOST_FULL
}
```

Initializing RAIL for `RAIL_DataMethod_t::FIFO_MODE` requires a few more function calls:

```
extern RAIL_Handle_t railHandle;

void config_rail_for_fifo_mode_operation(void)
{
    RAIL_DataConfig_t railDataConfig = {
        .txSource = TX_PACKET_DATA,
        .rxSource = RX_PACKET_DATA,
        .txMethod = FIFO_MODE,
        .rxMethod = FIFO_MODE,
    };

    (void) RAIL_ConfigData(railHandle, &railDataConfig);

    // Gets the size of the FIFOs.
    // Assume that the transmit and receive FIFOs are the same size
    uint16_t fifoSize = RAIL_GetTxFifoSpaceAvailable(railHandle);

    // Sets the transmit and receive FIFO thresholds.
    // For this example, set the threshold in the middle of each FIFO.
    (void) RAIL_SetRxFifoThreshold(railHandle, fifoSize / 2);
    (void) RAIL_SetTxFifoThreshold(railHandle, fifoSize / 2);

    // Events that can occur in FIFO mode:
    // RAIL_EVENT_TX_FIFO_ALMOST_EMPTY
    // RAIL_EVENT_TX_UNDERFLOW
    // RAIL_EVENT_TXACK_UNDERFLOW
    // RAIL_EVENT_TX_PACKET_SENT
    // RAIL_EVENT_RX_FIFO_ALMOST_FULL
    // RAIL_EVENT_RX_FIFO_OVERFLOW
    // RAIL_EVENT_RX_ADDRESS_FILTERED
    // RAIL_EVENT_RX_PACKET_ABORTED
    // RAIL_EVENT_RX_FRAME_ERROR
    // RAIL_EVENT_RX_PACKET_RECEIVED
}
```

On receive, an application can use a different [RAIL_RxDataSource_t](#) that is only compatible with [RAIL_DataMethod_t::FIFO_MODE](#). All that differs from the FIFO mode example above is the [RAIL_DataConfig_t::rxSource](#) setting. IQ data samples are taken at the hardware's oversample rate and the amount of data can easily overwhelm the CPU processing time. The sample rate depends on the chosen PHY, as determined by the data rate and the decimation chain. It is **not** recommended to use the IQ data source with sample rates above 300 k samples/second because the CPU might not be able to keep up with the data stream. Depending on the application and the needed CPU bandwidth, slower data rates may be required. On EFR32xG22 and later platforms, it is recommended to reset the RX buffer before initiating a receive for all modes except [RAIL_RxDataSource_t::RX_PACKET_DATA](#) since the RX buffer has to be 32-bit aligned. If the buffer is **not** reset but is 32-bit aligned, capture is performed on the remaining space available. If the buffer is **not** reset and is **not** 32-bit aligned, then [RAIL_ConfigData\(\)](#) returns [RAIL_STATUS_INVALID_STATE](#).

```
// Reset RX buffer (EFR32xG22 and later platforms)
RAIL_ResetFifo(railHandle, false, true);

// IQ data is provided into the receive FIFO.
RAIL_DataConfig_t railDataConfig = {
    .txSource = TX_PACKET_DATA,
    .rxSource = RX_IQDATA_FILT_LSB,
    .txMethod = FIFO_MODE,
    .rxMethod = FIFO_MODE,
};

// IQ data comes in the following format when reading out of the FIFO:
//-----
// I[LSB] | I[MSB] | Q[LSB] | Q[MSB] |
//-----
```

Modules

[RAIL_DataConfig_t](#)

Enumerations

```
enum    RAIL_TxDataSource_t {
    TX_PACKET_DATA = 0
    TX_MFM_DATA = 1
    RAIL_TX_DATA_SOURCE_COUNT
}
Transmit data sources supported by RAIL.

enum    RAIL_RxDataSource_t {
    RX_PACKET_DATA = 0
    RX_DEMOD_DATA = 1
    RX_IQDATA_FILT_LSB = 2
    RX_IQDATA_FILT_MSB = 3
    RX_DIRECT_MODE_DATA = 4
    RX_DIRECT_SYNCHRONOUS_MODE_DATA = 5
    RAIL_RX_DATA_SOURCE_COUNT
}
Receive data sources supported by RAIL.

enum    RAIL_DataMethod_t {
    PACKET_MODE = 0
    FIFO_MODE = 1
    RAIL_DATA_METHOD_COUNT
}
Methods for the application to provide and retrieve data from RAIL.
```

Functions

RAIL_Status_t	RAIL_ConfigData (RAIL_Handle_t railHandle, const RAIL_DataConfig_t *dataConfig) RAIL data management configuration.
uint16_t	RAIL_WriteTxFifo (RAIL_Handle_t railHandle, const uint8_t *dataPtr, uint16_t writeLength, bool reset) Write data to the transmit FIFO previously established by RAIL_SetTxFifo() .
uint16_t	RAIL_SetTxFifo (RAIL_Handle_t railHandle, uint8_t *addr, uint16_t initLength, uint16_t size) Set the address of the transmit FIFO, a circular buffer used for TX data, possibly pre-populated with transmit data.
uint16_t	RAIL_SetTxFifoAlt (RAIL_Handle_t railHandle, uint8_t *addr, uint16_t startOffset, uint16_t initLength, uint16_t size) Set the address of the transmit FIFO, a circular buffer used for TX data which can start at offset distance from the FIFO base address.
RAIL_Status_t	RAIL_SetRxFifo (RAIL_Handle_t railHandle, uint8_t *addr, uint16_t *size) Set the address of the receive FIFO, a circular buffer used for receive data.
RAIL_Status_t	RAILCb_SetupRxFifo (RAIL_Handle_t railHandle) Set up the receive FIFO to use.
uint16_t	RAIL_ReadRxFifo (RAIL_Handle_t railHandle, uint8_t *dataPtr, uint16_t readLength) Read packet data from RAIL's receive FIFO.
uint16_t	RAIL_SetTxFifoThreshold (RAIL_Handle_t railHandle, uint16_t txThreshold) Configure the RAIL transmit FIFO almost empty threshold.
uint16_t	RAIL_SetRxFifoThreshold (RAIL_Handle_t railHandle, uint16_t rxThreshold) Configure the RAIL receive FIFO almost full threshold.
uint16_t	RAIL_GetTxFifoThreshold (RAIL_Handle_t railHandle) Get the RAIL transmit FIFO almost empty threshold value.
uint16_t	RAIL_GetRxFifoThreshold (RAIL_Handle_t railHandle) Get the RAIL receive FIFO almost full threshold value.
RAIL_Status_t	RAIL_ResetFifo (RAIL_Handle_t railHandle, bool txFifo, bool rxFifo) Reset the RAIL transmit and/or receive FIFOs.
uint16_t	RAIL_GetRxFifoBytesAvailable (RAIL_Handle_t railHandle) Get the number of bytes used in the receive FIFO.
uint16_t	RAIL_GetTxFifoSpaceAvailable (RAIL_Handle_t railHandle) Get the number of bytes unused in the transmit FIFO.

Macros

#define	RAIL_FIFO_ALIGNMENT_TYPE uint32_t Fixed-width type indicating the needed alignment for RX and TX FIFOs.
#define	RAIL_FIFO_ALIGNMENT (sizeof(RAIL_FIFO_ALIGNMENT_TYPE)) Alignment that is needed for the RX and TX FIFOs.
#define	RAIL_FIFO_THRESHOLD_DISABLED 0xFFFFU A FIFO threshold value that disables the threshold.

Enumeration Documentation

RAIL_TxDataSource_t

RAIL_TxDataSource_t

Transmit data sources supported by RAIL.

Enumerator	
TX_PACKET_DATA	Uses the frame hardware to packetize data.
TX_MFM_DATA	Uses the multi-level frequency modulation data.
RAIL_TX_DATA_SOURCE_COUNT	A count of the choices in this enumeration.

Definition at line 2529 of file common/rail_types.h

RAIL_RxDataSource_t

RAIL_RxDataSource_t

Receive data sources supported by RAIL.

Note

- Data sources other than [RX_PACKET_DATA](#) require use of [RAIL_DataMethod_t::FIFO_MODE](#).

Enumerator	
RX_PACKET_DATA	Uses the frame hardware to packetize data.
RX_DEMOD_DATA	Gets 8-bit data output from the demodulator.
RX_IQDATA_FILTLSB	Gets lower 16 bits of I/Q data provided to the demodulator.
RX_IQDATA_FILTMSB	Gets highest 16 bits of I/Q data provided to the demodulator.
RX_DIRECT_MODE_DATA	Gets RX direct mode data output from the demodulator.
RX_DIRECT_SYNCHRONOUS_MODE_DATA	Gets synchronous RX direct mode data output from the demodulator.
RAIL_RX_DATA_SOURCE_COUNT	A count of the choices in this enumeration.

Definition at line 2558 of file common/rail_types.h

RAIL_DataMethod_t

RAIL_DataMethod_t

Methods for the application to provide and retrieve data from RAIL.

For Transmit the distinction between [RAIL_DataMethod_t::PACKET_MODE](#) and [RAIL_DataMethod_t::FIFO_MODE](#) has become more cosmetic than functional, as the [RAIL_WriteTxFifo\(\)](#) and [RAIL_SetTxFifoThreshold\(\)](#) APIs and related [RAIL_EVENT_TX_FIFO_ALMOST_EMPTY](#) event can be used in either mode. For Receive the distinction is functionally important because in [RAIL_DataMethod_t::PACKET_MODE](#) rollback occurs automatically for unsuccessfully-received packets ([RAIL_RxPacketStatus_t](#) ABORT statuses), flushing their data. In [RAIL_DataMethod_t::FIFO_MODE](#) rollback is prevented, leaving the data from unsuccessfully-received packets in the receive FIFO for the application to deal with. This allows for packets larger than the receive FIFO size where automatic rollback would corrupt the receive FIFO.

Enumerator	
PACKET_MODE	Packet-based data method.
FIFO_MODE	FIFO-based data method.
RAIL_DATA_METHOD_COUNT	A count of the choices in this enumeration.

Definition at line 2615 of file common/rail_types.h

Function Documentation

RAIL_ConfigData

```
RAIL_Status_t RAIL_ConfigData (RAIL_Handle_t railHandle, const RAIL_DataConfig_t * dataConfig)
```

RAIL data management configuration.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	dataConfig	A pointer to a non-NULL RAIL data configuration structure.

Returns

- Status code indicating success of the function call.

This function configures how RAIL manages data. The application can configure RAIL to receive data in a packet-based or FIFO-based manner. [RAIL_DataMethod_t::FIFO_MODE](#) is necessary to receive packets larger than the radio's receive FIFO. It is also required for receive data sources other than [RAIL_RxDataSource_t::RX_PACKET_DATA](#).

Generally with [RAIL_DataMethod_t::FIFO_MODE](#), the application sets appropriate FIFO thresholds via [RAIL_SetTxFifoThreshold\(\)](#) and [RAIL_SetRxFifoThreshold\(\)](#) and then enables and handles the [RAIL_EVENT_TX_FIFO_ALMOST_EMPTY](#) event callback (to feed more packet data via [RAIL_WriteTxFifo\(\)](#) before the FIFO underflows) and the [RAIL_EVENT_RX_FIFO_ALMOST_FULL](#) event callback (to consume packet data via [RAIL_ReadRxFifo\(\)](#) before the receive FIFO overflows).

When configuring TX for [RAIL_DataMethod_t::FIFO_MODE](#), this function resets the transmit FIFO. When configuring TX or RX for [RAIL_DataMethod_t::PACKET_MODE](#), this function will reset the corresponding FIFO thresholds such that they won't trigger the [RAIL_EVENT_RX_FIFO_ALMOST_FULL](#) or [RAIL_EVENT_TX_FIFO_ALMOST_EMPTY](#) events.

When [RAIL_DataConfig_t::rxMethod](#) is set to [RAIL_DataMethod_t::FIFO_MODE](#), the radio won't drop packet data of aborted or CRC error packets, but will present it to the application to deal with accordingly. On completion of erroneous packets, the [RAIL_Config_t::eventsCallback](#) with [RAIL_EVENT_RX_PACKET_ABORTED](#), [RAIL_EVENT_RX_FRAME_ERROR](#), or [RAIL_EVENT_RX_ADDRESS_FILTERED](#) will tell the application it can drop any data it read via [RAIL_ReadRxFifo\(\)](#) during reception. For CRC error packets when the [RAIL_RX_OPTION_IGNORE_CRC_ERRORS](#) RX option is in effect, the application should check for that from the [RAIL_RxPacketStatus_t](#) obtained by calling [RAIL_GetRxPacketInfo\(\)](#). RAIL will automatically flush any remaining packet data after reporting one of these packet completion events or the application can explicitly flush it by calling [RAIL_ReleaseRxPacket\(\)](#).

When [RAIL_DataConfig_t::rxMethod](#) is set to [RAIL_DataMethod_t::PACKET_MODE](#), the radio will roll back (drop) all packet data associated with aborted packets including those with CRC errors (unless configured to ignore CRC errors via the [RAIL_RX_OPTION_IGNORE_CRC_ERRORS](#) RX option). The application will never have to deal with packet data from these packets. In either mode, the application can set RX options as needed.

When [RAIL_DataConfig_t::rxSource](#) is set to a value other than [RX_PACKET_DATA](#) and [RAIL_Config_t::eventsCallbackRAIL_EVENT_RX_FIFO_OVERFLOW](#) is enabled RX will be terminated if a RX FIFO overflow occurs. If [RAIL_EVENT_RX_FIFO_OVERFLOW](#) is not enabled, data will be discarded until the overflow condition is resolved. To continue capturing data RX must be restarted using [RAIL_StartRx\(\)](#).

Definition at line 2004 of file `common/rail.h`

RAIL_WriteTxFifo

```
uint16_t RAIL_WriteTxFifo (RAIL_Handle_t railHandle, const uint8_t * dataPtr, uint16_t writeLength, bool reset)
```

Write data to the transmit FIFO previously established by [RAIL_SetTxFifo\(\)](#).

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	dataPtr	A pointer to transmit data.
[in]	writeLength	A number of bytes to write to the transmit FIFO.
[in]	reset	If true, resets transmit FIFO before writing the data.

Returns

- The number of bytes written to the transmit FIFO.

This function copies writeLength bytes of data from the provided dataPtr into the transmit FIFO previously established by [RAIL_SetTxFifo\(\)](#) or [RAIL_Init\(\)](#). If the requested writeLength exceeds the current number of bytes open in the transmit FIFO, the function only writes until the transmit FIFO is full. The function returns the number of bytes written to the transmit FIFO or returns zero if railHandle is NULL or if the transmit FIFO is full.

Note

- The protocol's packet configuration, as set up by the radio configurator or via [RAIL_SetFixedLength\(\)](#), determines how many bytes of data are consumed from the transmit FIFO for a successful transmit operation, not the writeLength value passed in. If not enough data has been put into the transmit FIFO, a [RAIL_EVENT_TX_UNDERFLOW](#) event will occur. If too much data is put into the transmit FIFO, the extra data will either become the first bytes sent in a subsequent packet, or will be thrown away if the FIFO gets reset prior to the next transmit. In general, the proper number of packet bytes to put into the transmit FIFO are all payload bytes except for any CRC bytes, which the packet configuration causes to be sent automatically.
- This function does not utilize a critical section but, depending on the application, calling it within a critical section could be appropriate.

Definition at line 2039 of file common/rail.h

RAIL_SetTxFifo

```
uint16_t RAIL_SetTxFifo (RAIL\_Handle\_t railHandle, uint8_t * addr, uint16_t initLength, uint16_t size)
```

Set the address of the transmit FIFO, a circular buffer used for TX data, possibly pre-populated with transmit data.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	addr	An appropriately-aligned (see below) pointer to a read-write memory location in RAM used as the transmit FIFO. This memory must persist until the next call to this function or RAIL_SetTxFifoAlt .
[in]	initLength	A number of initial bytes already in the transmit FIFO.
[in]	size	A desired size of the transmit FIFO in bytes.

Returns

- The transmit FIFO size in bytes, 0 if an error occurs.

This function sets the memory location for the transmit FIFO. [RAIL_SetTxFifo](#) or [RAIL_SetTxFifoAlt](#) must be called at least once before any transmit operations occur.

FIFO size can be determined by the return value of this function. The chosen size is determined based on the available FIFO sizes supported by the hardware. Similarly, some hardware has stricter FIFO alignment requirements; 32-bit alignment provides the maximum portability across all RAIL platforms. For more on supported FIFO sizes and alignments, see chip-specific documentation, such as [EFR32](#). The returned FIFO size will be the closest allowed size less than or equal to the passed in size parameter, unless the size parameter is smaller than the minimum FIFO size, in that case 0 is returned. If the initLength parameter is larger than the returned size, the FIFO will be filled up to its size.

A user may write to the custom memory location directly before calling this function, or use [RAIL_WriteTxFifo\(\)](#) to write to the memory location after calling this function. Users must specify the initLength for previously-written memory to be set in

the transmit FIFO.

This function reserves the block of RAM starting at `addr` with a length of the returned FIFO size, which is used internally as a circular buffer for the transmit FIFO. It must be able to hold the entire FIFO size. The caller must guarantee that the custom FIFO remains intact and unchanged (except via calls to `RAIL_WriteTxFifo()`) until the next call to this function.

Note

- The protocol's packet configuration, as set up by the radio configurator or via `RAIL_SetFixedLength()`, determines how many bytes of data are consumed from the transmit FIFO for a successful transmit operation, not the `initLength` value passed in. If not enough data has been put into the transmit FIFO, a `RAIL_EVENT_TX_UNDERFLOW` event will occur. If too much data is put into the transmit FIFO, the extra data will either become the first bytes sent in a subsequent packet, or will be thrown away if the FIFO gets reset prior to the next transmit. In general, the proper number of packet bytes to put into the transmit FIFO are all payload bytes except for any CRC bytes which the packet configuration causes to be sent automatically.

Definition at line 2094 of file `common/rail.h`

RAIL_SetTxFifoAlt

```
uint16_t RAIL_SetTxFifoAlt (RAIL_Handle_t railHandle, uint8_t * addr, uint16_t startOffset, uint16_t initLength, uint16_t size)
```

Set the address of the transmit FIFO, a circular buffer used for TX data which can start at offset distance from the FIFO base address.

Parameters

[in]	<code>railHandle</code>	A RAIL instance handle.
[inout]	<code>addr</code>	An appropriately-aligned (see <code>RAIL_SetTxFifo</code> description) pointer to a read-write memory location in RAM used as the transmit FIFO. This memory must persist until the next call to this function or <code>RAIL_SetTxFifo</code> .
[in]	<code>startOffset</code>	A number of bytes defining the start position of the TX data from the transmit FIFO base address, only valid if <code>initLength</code> is not 0.
[in]	<code>initLength</code>	The number of valid bytes already in the transmit FIFO after <code>startOffset</code> .
[in]	<code>size</code>	A desired size of the transmit FIFO in bytes.

Returns

- The transmit FIFO size in bytes, 0 if an error occurs.

This function is similar to `RAIL_SetTxFifo` except a `startOffset` can be specified to indicate where the transmit packet data starts. This allows an application to place unaligned initial packet data within the aligned transmit FIFO (`initLength > 0`). Specifying a `startOffset` will not reduce the FIFO threshold or affect `RAIL_GetTxFifoSpaceAvailable()`. `RAIL_SetTxFifo` or `RAIL_SetTxFifoAlt` must be called at least once before any transmit operations occur. FIFO size handling is quite same as `RAIL_SetTxFifo`. Only difference is that if the `initLength` plus `startOffset` parameters are larger than the returned size, the FIFO will be filled up to its size from `startOffset`. Note that the `startOffset` is essentially forgotten after the next transmit – i.e. it applies onto to the next transmit operation, and is not re-established when the transmit FIFO is reset.

Definition at line 2127 of file `common/rail.h`

RAIL_SetRxFifo

```
RAIL_Status_t RAIL_SetRxFifo (RAIL_Handle_t railHandle, uint8_t * addr, uint16_t * size)
```

Set the address of the receive FIFO, a circular buffer used for receive data.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	addr	A pointer to a read-write memory location in RAM used as the receive FIFO. This memory must persist until the next call to this function.
[inout]	size	A pointer to the desired size of the receive FIFO in bytes. This will be updated with the actual size during the function call.

Returns

- Status code indicating success of the function call.

This function sets the memory location for the receive FIFO. It must be called at least once before any receive operations occur.

Note

- After it is called, any prior receive FIFO is orphaned. To avoid orphaning the default internal 512-byte receive FIFO so it does not unnecessarily consume RAM resources in your application, implement [RAILCb_SetupRxFifo\(\)](#) to call this function.

FIFO size can be determined by the return value of this function. The chosen size is determined based on the available FIFO sizes supported by the hardware. Similarly, some hardware has stricter FIFO alignment requirements; 32-bit alignment provides the maximum portability across all RAIL platforms. For more on supported FIFO sizes and alignments, see chip-specific documentation, such as [EFR32](#). The returned FIFO size will be the closest allowed size less than or equal to the passed in size parameter, unless the size parameter is smaller than the minimum FIFO size.

This function reserves the block of RAM starting at `addr` with a length of `size`, which is used internally as a circular buffer for the receive FIFO. It must be able to hold the entire FIFO size. The caller must guarantee that the custom FIFO remains intact and unchanged (except via incoming packet data being written) until the next call to this function.

In multiprotocol, RAIL currently shares one receive FIFO across all protocols. This function will return [RAIL_STATUS_INVALID_STATE](#) if the requested [RAIL_Handle_t](#) is not active.

Definition at line 2174 of file `common/rail.h`

RAILCb_SetupRxFifo

```
RAIL_Status_t RAILCb_SetupRxFifo (RAIL_Handle_t railHandle)
```

Set up the receive FIFO to use.

Parameters

[in]	railHandle	A RAIL instance handle.

This function is optional to implement.

Returns

- Status code indicating success of the function call.

This callback is called during the [RAIL_Init\(\)](#) process to set up the FIFO to use for received packets. If not implemented by the application, a default implementation from within the RAIL library will be used to initialize an internal default 512-byte receive FIFO.

If this function returns an error, the [RAIL_Init\(\)](#) process will fail.

During this function, the application should generally call [RAIL_SetRxFifo\(\)](#). If that does not happen, the application needs to set up the receive FIFO via a call to [RAIL_SetRxFifo\(\)](#) before attempting to receive any packets. An example implementation may look like the following:

```
#define RX_FIFO_BYTES 1024
static __ALIGNED(RAIL_FIFO_ALIGNMENT) uint8_t rxFifo[RX_FIFO_BYTES];
```

```
RAIL_Status_t RAILCb_SetupRxFifo(RAIL_Handle_t railHandle){
    uint16_t rxFifoBytes = RX_FIFO_BYTES;
    RAIL_Status_t status = RAIL_SetRxFifo(railHandle,&rxFifo[0],&rxFifoBytes);if(rxFifoBytes != RX_FIFO_BYTES){// We set up an incorrect FIFO
size}return RAIL_STATUS_INVALID_PARAMETER;}if(status == RAIL_STATUS_INVALID_STATE){// Allow failures due to multiprotocolreturn
RAIL_STATUS_NO_ERROR;}return status;}
```

Definition at line 2213 of file common/rail.h

RAIL_ReadRxFifo

```
uint16_t RAIL_ReadRxFifo (RAIL_Handle_t railHandle, uint8_t * dataPtr, uint16_t readLength)
```

Read packet data from RAIL's receive FIFO.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	dataPtr	An application-provided pointer to store data. If NULL, the data is thrown away rather than copied out.
[in]	readLength	A number of packet bytes to read from the FIFO.

Returns

- The number of packet bytes read from the receive FIFO.
- This function reads packet data from the head of receive FIFO and writes it to the provided dataPtr. It does not permit reading more data than is available in the FIFO, nor does it permit reading more data than remains in the oldest unreleased packet.
- Because this function does not have a critical section, use it only in one context or make sure function calls are protected to prevent buffer corruption.

Warnings

- This function is intended for use only with [RAIL_DataMethod_t::FIFO_MODE](#) and should never be called in [RAIL_DataMethod_t::PACKET_MODE](#) where it could lead to receive FIFO corruption.

Note

- When reading data from an arriving packet that is not yet complete, keep in mind its data is highly suspect because it has not yet passed any CRC integrity checking. Also note that the packet could be aborted, canceled, or fail momentarily, invalidating its data in Packet mode. Furthermore, there is a small chance towards the end of packet reception that the receive FIFO could include not only packet data received so far, but also some raw radio-appended info detail bytes that RAIL's packet-completion processing will subsequently deal with. It's up to the application to know its packet format well enough to avoid reading this info because it will corrupt the packet's details and possibly corrupt the receive FIFO.

Definition at line 2252 of file common/rail.h

RAIL_SetTxFifoThreshold

```
uint16_t RAIL_SetTxFifoThreshold (RAIL_Handle_t railHandle, uint16_t txThreshold)
```

Configure the RAIL transmit FIFO almost empty threshold.

Parameters

[in]	railHandle	A RAIL instance handle.

[in]	txThreshold	The threshold below which the RAIL_EVENT_TX_FIFO_ALMOST_EMPTY event will fire.

Returns

- Configured transmit FIFO threshold value.

This function configures the threshold for the transmit FIFO. When the number of bytes in the transmit FIFO falls below the configured threshold, [RAIL_Config_t::eventsCallback](#) will fire with [RAIL_EVENT_TX_FIFO_ALMOST_EMPTY](#) set. The txThreshold value should be smaller than or equal to the transmit FIFO size; higher values will be pegged to the FIFO size. A value of 0 or [RAIL_FIFO_THRESHOLD_DISABLED](#) will disable the threshold, returning [RAIL_FIFO_THRESHOLD_DISABLED](#).

Definition at line 2273 of file `common/rail.h`

RAIL_SetRxFifoThreshold

```
uint16_t RAIL_SetRxFifoThreshold (RAIL\_Handle\_t railHandle, uint16_t rxThreshold)
```

Configure the RAIL receive FIFO almost full threshold.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	rxThreshold	The threshold above which the RAIL_EVENT_RX_FIFO_ALMOST_FULL event will fire.

Returns

- Configured receive FIFO threshold value.

This function configures the threshold for the receive FIFO. When the number of bytes of packet data in the receive FIFO exceeds the configured threshold, [RAIL_Config_t::eventsCallback](#) will keep firing with [RAIL_EVENT_RX_FIFO_ALMOST_FULL](#) set as long as the number of bytes in the receive FIFO exceeds the configured threshold value. The rxThreshold value should be smaller than the receive FIFO size; anything else, including a value of [RAIL_FIFO_THRESHOLD_DISABLED](#), will disable the threshold, returning [RAIL_FIFO_THRESHOLD_DISABLED](#).

Note

- To avoid sticking in the event handler (even in idle state):
 - Disable the event (via the config events API or the [RAIL_FIFO_THRESHOLD_DISABLED](#) parameter).
 - Increase FIFO threshold.
 - Read the FIFO (that's not an option in [RAIL_DataMethod_t::PACKET_MODE](#)) in the event handler.

Definition at line 2301 of file `common/rail.h`

RAIL_GetTxFifoThreshold

```
uint16_t RAIL_GetTxFifoThreshold (RAIL\_Handle\_t railHandle)
```

Get the RAIL transmit FIFO almost empty threshold value.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Configured TX Threshold value in bytes.

Retrieves the configured TX threshold value.

Definition at line 2312 of file common/rail.h

RAIL_GetRxFifoThreshold

```
uint16_t RAIL_GetRxFifoThreshold (RAIL_Handle_t railHandle)
```

Get the RAIL receive FIFO almost full threshold value.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Configured RX Threshold value in bytes.
- Retrieves the configured RX threshold value.

Definition at line 2322 of file common/rail.h

RAIL_ResetFifo

```
RAIL_Status_t RAIL_ResetFifo (RAIL_Handle_t railHandle, bool txFifo, bool rxFifo)
```

Reset the RAIL transmit and/or receive FIFOs.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	txFifo	If true, reset the transmit FIFO.
[in]	rxFifo	If true, reset the receive FIFO and its internal metadata FIFO.

Returns

- Status code indicating success of the function call.
- This function can reset each FIFO independently. The application should not reset the receive FIFO while receiving a frame, nor should it reset the transmit FIFO while transmitting a frame.

Definition at line 2336 of file common/rail.h

RAIL_GetRxFifoBytesAvailable

```
uint16_t RAIL_GetRxFifoBytesAvailable (RAIL_Handle_t railHandle)
```

Get the number of bytes used in the receive FIFO.

Parameters

[in]	railHandle	A RAIL instance handle.

Only use this function in RX [RAIL_DataMethod_t::FIFO_MODE](#). Apps should use [RAIL_GetRxPacketInfo\(\)](#) instead.

Returns

- Number of bytes used in the receive FIFO.

This function indicates how much packet-related data exists in the receive FIFO that could be read.

Note

- The number of bytes returned may not just reflect the current packet's data but could also include raw appended info bytes added after successful packet reception and bytes from subsequently received packets. It is up to the app to never try to consume more than the packet's actual data when using the value returned here in a subsequent call to [RAIL_ReadRxFifo\(\)](#), otherwise the receive FIFO will be corrupted.

Definition at line 2356 of file common/rail.h

RAIL_GetTxFifoSpaceAvailable

```
uint16_t RAIL_GetTxFifoSpaceAvailable (RAIL_Handle_t railHandle)
```

Get the number of bytes unused in the transmit FIFO.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Number of bytes unused in the transmit FIFO.

This function indicates how much space is available in the transmit FIFO for writing additional packet data.

Definition at line 2367 of file common/rail.h

RAIL_DataConfig_t

RAIL data configuration structure.

Select the transmit/receive data sources and the method the application uses to provide/retrieve data from RAIL.

Public Attributes

RAIL_TxDataSource_t	txSource Source of TX Data.
RAIL_RxDataSource_t	rxSource Source of RX Data.
RAIL_DataMethod_t	txMethod Method of providing transmit data.
RAIL_DataMethod_t	rxMethod Method of retrieving receive data.

Public Attribute Documentation

txSource

```
RAIL_TxDataSource_t RAIL_DataConfig_t::txSource
```

Source of TX Data.

Definition at line 2646 of file common/rail_types.h

rxSource

```
RAIL_RxDataSource_t RAIL_DataConfig_t::rxSource
```

Source of RX Data.

Definition at line 2648 of file common/rail_types.h

txMethod

```
RAIL_DataMethod_t RAIL_DataConfig_t::txMethod
```

Method of providing transmit data.

Definition at line 2650 of file common/rail_types.h

rxMethod

```
RAIL_DataMethod_t RAIL_DataConfig_t::rxMethod
```

Method of retrieving receive data.

Definition at line 2652 of file common/rail_types.h

Diagnostic

Diagnostic

APIs for diagnostic and test chip modes.

Modules

- [RAIL_DirectModeConfig_t](#)
- [RAIL_VerifyConfig_t](#)

Enumerations

```
enum RAIL_StreamMode_t {  
    RAIL_STREAM_CARRIER_WAVE = 0  
    RAIL_STREAM_PN9_STREAM = 1  
    RAIL_STREAM_10_STREAM = 2  
    RAIL_STREAM_CARRIER_WAVE_PHASENOISE = 3  
    RAIL_STREAM_RAMP_STREAM = 4  
    RAIL_STREAM_CARRIER_WAVE_SHIFTED = 5  
    RAIL_STREAM_1000_STREAM = 6  
    RAIL_STREAM_MODES_COUNT  
}  
Possible stream output modes.
```

Typedefs

```
typedef int16_t RAIL_FrequencyOffset_t  
Type that represents the number of Frequency Offset units.  
  
typedef bool(* RAIL_VerifyCallbackPtr_t)(uint32_t address, uint32_t expectedValue, uint32_t actualValue)  
A pointer to a verification callback function.
```

Functions

RAIL_Status_t	RAIL_ConfigDirectMode (RAIL_Handle_t railHandle, const RAIL_DirectModeConfig_t *directModeConfig) Configure direct mode for RAIL.
RAIL_Status_t	RAIL_EnableDirectMode (RAIL_Handle_t railHandle, bool enable) Enable or disable direct mode for RAIL.
RAIL_Status_t	RAIL_EnableDirectModeAlt (RAIL_Handle_t railHandle, bool enableDirectTx, bool enableDirectRx) Enable or disable direct mode for RAIL.
uint32_t	RAIL_GetRadioClockFreqHz (RAIL_Handle_t railHandle) Get the radio subsystem clock frequency in Hz.
RAIL_Status_t	RAIL_SetTune (RAIL_Handle_t railHandle, uint32_t tune) Set the crystal tuning.
uint32_t	RAIL_GetTune (RAIL_Handle_t railHandle) Get the crystal tuning.
RAIL_Status_t	RAIL_SetTuneDelta (RAIL_Handle_t railHandle, int32_t delta)

Set the crystal tuning delta.

int32_t	RAIL_GetTuneDelta (RAIL_Handle_t railHandle) Get the crystal tuning delta.
RAIL_FrequencyOffset_t	RAIL_GetRxFreqOffset (RAIL_Handle_t railHandle) Get the frequency offset.
RAIL_Status_t	RAIL_SetFreqOffset (RAIL_Handle_t railHandle, RAIL_FrequencyOffset_t freqOffset) Set the nominal radio frequency offset.
RAIL_Status_t	RAIL_StartTxStream (RAIL_Handle_t railHandle, uint16_t channel, RAIL_StreamMode_t mode) Start transmitting a stream on a certain channel.
RAIL_Status_t	RAIL_StartTxStreamAlt (RAIL_Handle_t railHandle, uint16_t channel, RAIL_StreamMode_t mode, RAIL_TxOptions_t options) Start transmitting a stream on a certain channel with the ability to select an antenna.
RAIL_Status_t	RAIL_StopTxStream (RAIL_Handle_t railHandle) Stop stream transmission and idle the radio.
RAIL_Status_t	RAIL_StopInfinitePreambleTx (RAIL_Handle_t railHandle) Stop infinite preamble transmission started and start transmitting the rest of the packet.
RAIL_Status_t	RAIL_ConfigVerification (RAIL_Handle_t railHandle, RAIL_VerifyConfig_t *configVerify, RAIL_RadioConfig_t radioConfig, RAIL_VerifyCallbackPtr_t cb) Configure the verification of radio memory contents.
RAIL_Status_t	RAIL_Verify (RAIL_VerifyConfig_t *configVerify, uint32_t durationUs, bool restart) Verify radio memory contents.

Macros

#define	RAIL_FREQUENCY_OFFSET_MAX ((RAIL_FrequencyOffset_t) 0x3FFF) The maximum frequency offset value supported.
#define	RAIL_FREQUENCY_OFFSET_MIN ((RAIL_FrequencyOffset_t) -RAIL_FREQUENCY_OFFSET_MAX) The minimum frequency offset value supported.
#define	RAIL_FREQUENCY_OFFSET_INVALID ((RAIL_FrequencyOffset_t) 0x8000) Specify an invalid frequency offset value.
#define	RAIL_VERIFY_DURATION_MAX 0xFFFFFFFFUL This radio state verification duration indicates to RAIL that all memory contents should be verified by RAIL before returning to the application.

Enumeration Documentation

RAIL_StreamMode_t

RAIL_StreamMode_t

Possible stream output modes.

Enumerator	
RAIL_STREAM_CARRIER_WAVE	An unmodulated carrier wave.
RAIL_STREAM_PN9_STREAM	PN9 byte sequence.
RAIL_STREAM_10_STREAM	101010 sequence.

RAIL_STREAM_CARRIER_WAVE_PHASENOISE	An unmodulated carrier wave with no change to PLL BW.
RAIL_STREAM_RAMP_STREAM	ramp sequence starting at a different offset for consecutive packets.
RAIL_STREAM_CARRIER_WAVE_SHIFTED	An unmodulated carrier wave not centered on DC but shifted roughly by channel_bandwidth/6 allowing an easy check of the residual DC.
RAIL_STREAM_1000_STREAM	10001000 sequence.
RAIL_STREAM_MODES_COUNT	A count of the choices in this enumeration.

Definition at line 5535 of file common/rail_types.h

Typedef Documentation

RAIL_FrequencyOffset_t

RAIL_FrequencyOffset_t

Type that represents the number of Frequency Offset units.

It is used with [RAIL_GetRxFreqOffset\(\)](#) and [RAIL_SetFreqOffset\(\)](#).

The units are chip-specific. For EFR32 they are radio synthesizer resolution steps (synth ticks) and is limited to 15 bits. A value of [RAIL_FREQUENCY_OFFSET_INVALID](#) means that this value is invalid.

Definition at line 5479 of file common/rail_types.h

RAIL_VerifyCallbackPtr_t

```
typedef bool(* RAIL_VerifyCallbackPtr_t) (uint32_t address, uint32_t expectedValue, uint32_t actualValue) (uint32_t address, uint32_t expectedValue, uint32_t actualValue)
```

A pointer to a verification callback function.

Parameters

[in]	address	The address of the data in question.
[in]	expectedValue	The expected value of the data in question.
[in]	actualValue	The actual value of the data in question.

This will be called by the radio state verification feature built into RAIL when a verified memory value is different from its reference value.

Returns

- true indicates a data value difference is acceptable. false indicates a data value difference in unacceptable.

Note

- This callback will be issued when an address' value is different from its reference value and either of the following conditions are met:
 1. The default radio configuration provided by the radio configurator is used for verification purposes (i.e., a custom radio configuration is not supplied as an input to [RAIL_ConfigVerification\(\)](#)), and the radio configurator has flagged the address under question as being verifiable.
 2. A custom radio configuration is provided to the verification API (i.e., a custom radio configuration is supplied as an input to [RAIL_ConfigVerification\(\)](#)). When providing a custom radio configuration for verification purposes, all addresses in that configuration will be verified, regardless of whether or not the addresses are flagged as verifiable.

Definition at line 5597 of file common/rail_types.h

Function Documentation

RAIL_ConfigDirectMode

RAIL_Status_t RAIL_ConfigDirectMode ([RAIL_Handle_t](#) railHandle, const [RAIL_DirectModeConfig_t](#) * directModeConfig)

Configure direct mode for RAIL.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	directModeConfig	A pointer to a configuration structure to specify direct mode parameters. Default configuration will be used if NULL is passed.

Returns

- [RAIL_STATUS_NO_ERROR](#) on success and an error code on failure.

This API configures direct mode and should be called before calling [RAIL_EnableDirectMode\(\)](#). If this function is not called, the following default [RAIL_DirectModeConfig_t](#) configuration will be used:

```
RAIL_DirectModeConfig_t defaultConfig = {
    .syncRx = false,
    .syncTx = false,
    .doutPort = SL_GPIO_PORT_A,
    .doutPin = 5,
    .dinPort = SL_GPIO_PORT_A,
    .dinPin = 7,
    .dclkPort = SL_GPIO_PORT_A,
    .dclkPin = 6,
};
```

Warnings

- This API is not safe to use in a multiprotocol app.

Definition at line 6048 of file common/rail.h

RAIL_EnableDirectMode

RAIL_Status_t RAIL_EnableDirectMode ([RAIL_Handle_t](#) railHandle, bool enable)

Enable or disable direct mode for RAIL.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	Whether or not to enable direct mode for TX and RX.

Returns

- [RAIL_STATUS_NO_ERROR](#) on success and an error code on failure.

See [RAIL_EnableDirectModeAlt\(\)](#) for more detailed function description.

Warnings

New applications should consider using [RAIL_EnableDirectModeAlt\(\)](#) for this functionality.

Note

- This feature is only available on certain devices. [RAIL_SupportsDirectMode\(\)](#) can be used to check if a particular device supports this feature or not.

Warnings

- As this function relies on GPIO access and RAIL is meant to run in TrustZone non-secure world, it is not supported if GPIO is configured as secure peripheral and it will return [RAIL_STATUS_INVALID_CALL](#).

Definition at line 6072 of file `common/rail.h`

RAIL_EnableDirectModeAlt

RAIL_Status_t RAIL_EnableDirectModeAlt ([RAIL_Handle_t](#) railHandle, bool enableDirectTx, bool enableDirectRx)

Enable or disable direct mode for RAIL.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enableDirectTx	Enable direct mode for data being transmitted out of the radio.
[in]	enableDirectRx	Enable direct mode for data being received from the radio.

Returns

- [RAIL_STATUS_NO_ERROR](#) on success or an error code on failure.

This API enables or disables the modem and GPIOs for direct mode operation. see [RAIL_ConfigDirectMode\(\)](#) for information on selecting the correct hardware configuration. If direct mode is enabled, packets are output and input directly to the radio via GPIO and RAIL packet handling is ignored.

Note

- This feature is only available on certain chips. [RAIL_SupportsDirectMode\(\)](#) can be used to check if a particular chip supports this feature or not.

Warnings

- This API is not safe to use in a multiprotocol app.
- As this function relies on GPIO access and RAIL is meant to run in TrustZone non-secure world, it is not supported if GPIO is configured as secure peripheral and it will return [RAIL_STATUS_INVALID_CALL](#).

Definition at line 6101 of file `common/rail.h`

RAIL_GetRadioClockFreqHz

uint32_t RAIL_GetRadioClockFreqHz ([RAIL_Handle_t](#) railHandle)

Get the radio subsystem clock frequency in Hz.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Radio subsystem clock frequency in Hz.

Definition at line 6111 of file common/rail.h

RAIL_SetTune

RAIL_Status_t RAIL_SetTune ([RAIL_Handle_t](#) railHandle, uint32_t tune)

Set the crystal tuning.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	tune	A chip-dependent crystal capacitor bank tuning parameter.

Returns

- Status code indicating success of the function call.

Tunes the crystal that the radio depends on to change the location of the center frequency for transmitting and receiving. This function will only succeed if the radio is idle at the time of the call.

Note

- This function proportionally affects the entire chip's timing across all its peripherals, including radio tuning and channel spacing. It is recommended to call this function only when HFXO is not being used, as it can cause disturbance on the HFXO frequency. A separate function, [RAIL_SetFreqOffset\(\)](#), can be used to adjust just the radio tuner without disturbing channel spacing or other chip peripheral timing.
- This API sets CTUNEXIANA and internally CTUNEXOANA = CTUNEXIANA + delta where delta is set or changed by [RAIL_SetTuneDelta\(\)](#). The default delta may not be 0 on some devices.

Definition at line 6135 of file common/rail.h

RAIL_GetTune

uint32_t RAIL_GetTune ([RAIL_Handle_t](#) railHandle)

Get the crystal tuning.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- A chip-dependent crystal capacitor bank tuning parameter.

Retrieves the current tuning value used by the crystal that the radio depends on. **Note**

- This is the CTUNEXIANA value.

Definition at line 6147 of file common/rail.h

RAIL_SetTuneDelta

RAIL_Status_t RAIL_SetTuneDelta ([RAIL_Handle_t](#) railHandle, int32_t delta)

Set the crystal tuning delta.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	delta	A chip-dependent crystal capacitor bank tuning delta.

Returns

- Status code indicating success of the function call.

Set the CTUNEXOANA delta for [RAIL_SetTune\(\)](#) to use: CTUNEXOANA = CTUNEXIANA + delta (subject to field-size limitations). This function does not change CTUNE values; call [RAIL_SetTune\(\)](#) to put a new delta into effect.

Definition at line 6161 of file common/rail.h

RAIL_GetTuneDelta

int32_t RAIL_GetTuneDelta ([RAIL_Handle_t](#) railHandle)

Get the crystal tuning delta.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- A chip-dependent crystal capacitor bank tuning delta.
- Retrieves the current tuning delta used by [RAIL_SetTune\(\)](#). **Note**
- The default delta if [RAIL_SetTuneDelta\(\)](#) has never been called is device-dependent and may not be 0.

Definition at line 6173 of file common/rail.h

RAIL_GetRxFreqOffset

RAIL_FrequencyOffset_t RAIL_GetRxFreqOffset ([RAIL_Handle_t](#) railHandle)

Get the frequency offset.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The measured frequency offset on a received packet. The units are described in the [RAIL_FrequencyOffset_t](#) documentation. If this returns [RAIL_FREQUENCY_OFFSET_INVALID](#), it was called while the radio wasn't active and there is no way to get the frequency offset.
- Retrieves the measured frequency offset used during the previous received packet, which includes the current radio frequency offset (see [RAIL_SetFreqOffset\(\)](#)). If the chip has not been in RX, it returns the nominal radio frequency offset.
- Note**
- Changing to any non-idle radio state after reception can cause this value to be overwritten so it is safest to capture during packet reception.

Definition at line 6193 of file common/rail.h

RAIL_SetFreqOffset

```
RAIL_Status_t RAIL_SetFreqOffset (RAIL_Handle_t railHandle, RAIL_FrequencyOffset_t freqOffset)
```

Set the nominal radio frequency offset.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	freqOffset	RAIL_FrequencyOffset_t parameter (signed, 2's complement).

Returns

- Status code indicating success of the function call.

This function is used to adjust the radio's tuning frequency slightly up or down. It might be used in conjunction with [RAIL_GetRxFreqOffset\(\)](#) after receiving a packet from a peer to adjust the tuner to better match the peer's tuned frequency.

Note

- Unlike [RAIL_SetTune\(\)](#), which affects the entire chip's timing including radio tuning and channel spacing, this function only affects radio tuning without disturbing channel spacing or other chip peripheral timing.

Definition at line 6213 of file `common/rail.h`

RAIL_StartTxStream

```
RAIL_Status_t RAIL_StartTxStream (RAIL_Handle_t railHandle, uint16_t channel, RAIL_StreamMode_t mode)
```

Start transmitting a stream on a certain channel.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	A channel on which to emit a stream.
[in]	mode	Choose the stream mode (PN9, and so on).

Returns

- Status code indicating success of the function call.

Begins streaming onto the given channel. The sources can either be an unmodulated carrier wave or an encoded stream of bits from a PN9 source. All ongoing radio operations will be stopped before transmission begins.

Definition at line 6228 of file `common/rail.h`

RAIL_StartTxStreamAlt

```
RAIL_Status_t RAIL_StartTxStreamAlt (RAIL_Handle_t railHandle, uint16_t channel, RAIL_StreamMode_t mode, RAIL_TxOptions_t options)
```

Start transmitting a stream on a certain channel with the ability to select an antenna.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	A channel on which to emit a stream.
[in]	mode	Choose the stream mode (PN9, and so on).
[in]	options	Choose the TX Antenna option. Takes only RAIL_TX_OPTION_ANTENNA0 , RAIL_TX_OPTION_ANTENNA1 , RAIL_TX_OPTIONS_DEFAULT or RAIL_TX_OPTIONS_NONE from the RAIL_TxOptions_t . If some other value is used then, transmission is possible on any antenna.

Returns

- Status code indicating success of the function call.

Begins streaming onto the given channel. The sources can either be an unmodulated carrier wave or an encoded stream of bits from a PN9 source. All ongoing radio operations will be stopped before transmission begins.

Definition at line 6250 of file `common/rail.h`

RAIL_StopTxStream

RAIL_Status_t RAIL_StopTxStream ([RAIL_Handle_t](#) railHandle)

Stop stream transmission and idle the radio.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

Halts the transmission started by [RAIL_StartTxStream\(\)](#).

Definition at line 6263 of file `common/rail.h`

RAIL_StopInfinitePreambleTx

RAIL_Status_t RAIL_StopInfinitePreambleTx ([RAIL_Handle_t](#) railHandle)

Stop infinite preamble transmission started and start transmitting the rest of the packet.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call: [RAIL_STATUS_NO_ERROR](#) if infinite preamble was stopped; [RAIL_STATUS_INVALID_CALL](#) if the radio isn't configured for infinite preamble; [RAIL_STATUS_INVALID_STATE](#) if the radio isn't transmitting.

This function is only useful for radio configurations that specify an infinite preamble. Call this API only after [RAIL_EVENT_TX_STARTED](#) has occurred and the radio is transmitting.

Definition at line 6280 of file `common/rail.h`

RAIL_ConfigVerification

RAIL_Status_t RAIL_ConfigVerification (RAIL_Handle_t railHandle, RAIL_VerifyConfig_t * configVerify, RAIL_RadioConfig_t radioConfig, RAIL_VerifyCallbackPtr_t cb)

Configure the verification of radio memory contents.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	configVerify	A pointer to a configuration structure made available to RAIL to perform radio state verification. This structure must be allocated in application global read-write memory. RAIL may modify fields within or referenced by this structure during its operation.
[in]	radioConfig	A radio configuration (pointer) that is to be used as a white list for verifying memory contents.
[in]	cb	A callback that notifies the application of a mismatch in expected vs actual memory contents. A NULL parameter may be passed in if a callback is not needed by the application.

Returns

- RAIL_STATUS_NO_ERROR if setup of the verification feature successfully occurred. RAIL_STATUS_INVALID_PARAMETER is returned if the provided railHandle or configVerify structures are invalid.

Definition at line 6300 of file common/rail.h

RAIL_Verify

RAIL_Status_t RAIL_Verify (RAIL_VerifyConfig_t * configVerify, uint32_t durationUs, bool restart)

Verify radio memory contents.

Parameters

[inout]	configVerify	A pointer to the configuration structure previously established by RAIL_ConfigVerification().
[in]	durationUs	The duration (in microseconds) for how long memory verification should occur before returning to the application. A value of RAIL_VERIFY_DURATION_MAX indicates that all memory contents should be verified before returning to the application.
[in]	restart	This flag only has meaning if a previous call of this function returned RAIL_STATUS_SUSPENDED. By restarting (true), the verification process starts over from the beginning, or by resuming where verification left off after being suspended (false), verification can proceed towards completion.

Returns

- RAIL_STATUS_NO_ERROR if the contents of all applicable memory locations have been verified. RAIL_STATUS_SUSPENDED is returned if the provided test duration expired but the time was not sufficient to verify all memory contents. By calling RAIL_Verify() again, further verification will commence. RAIL_STATUS_INVALID_PARAMETER is returned if the provided verifyConfig structure pointer is not configured for use by the active RAIL handle. RAIL_STATUS_INVALID_STATE is returned if any of the verified memory contents are different from their reference values.

Definition at line 6330 of file common/rail.h

RAIL_DirectModeConfig_t

Allows the user to specify direct mode parameters using [RAIL_ConfigDirectMode\(\)](#).

Public Attributes

bool	syncRx	Enable synchronous RX DOUT using DCLK vs.
bool	syncTx	Enable synchronous TX DIN using DCLK vs.
uint8_t	doutPort	RX Data output (DOUT) GPIO port.
uint8_t	doutPin	RX Data output (DOUT) GPIO pin.
uint8_t	dclkPort	Data clock (DCLK) GPIO port.
uint8_t	dclkPin	Data clock (DCLK) GPIO pin.
uint8_t	dinPort	TX Data input (DIN) GPIO port.
uint8_t	dinPin	TX Data input (DIN) GPIO pin.
uint8_t	doutLoc	
uint8_t	dclkLoc	
uint8_t	dinLoc	

Public Attribute Documentation

syncRx

bool RAIL_DirectModeConfig_t::syncRx

Enable synchronous RX DOUT using DCLK vs.
asynchronous RX DOUT.

Definition at line 5504 of file `common/rail_types.h`

syncTx

bool RAIL_DirectModeConfig_t::syncTx

Enable synchronous TX DIN using DCLK vs.
asynchronous TX DIN.

Definition at line 5506 of file common/rail_types.h

doutPort

uint8_t RAIL_DirectModeConfig_t::doutPort

RX Data output (DOUT) GPIO port.

Definition at line 5509 of file common/rail_types.h

doutPin

uint8_t RAIL_DirectModeConfig_t::doutPin

RX Data output (DOUT) GPIO pin.

Definition at line 5511 of file common/rail_types.h

dclkPort

uint8_t RAIL_DirectModeConfig_t::dclkPort

Data clock (DCLK) GPIO port.

Only used in synchronous mode.

Definition at line 5514 of file common/rail_types.h

dclkPin

uint8_t RAIL_DirectModeConfig_t::dclkPin

Data clock (DCLK) GPIO pin.

Only used in synchronous mode.

Definition at line 5516 of file common/rail_types.h

dinPort

uint8_t RAIL_DirectModeConfig_t::dinPort

TX Data input (DIN) GPIO port.

Definition at line 5519 of file common/rail_types.h

dinPin

```
uint8_t RAIL_DirectModeConfig_t::dinPin
```

TX Data input (DIN) GPIO pin.

Definition at line 5521 of file common/rail_types.h

doutLoc

```
uint8_t RAIL_DirectModeConfig_t::doutLoc
```

DeprecatedNo longer used (ignored).

Definition at line 5524 of file common/rail_types.h

dclkLoc

```
uint8_t RAIL_DirectModeConfig_t::dclkLoc
```

DeprecatedNo longer used (ignored).

Definition at line 5526 of file common/rail_types.h

dinLoc

```
uint8_t RAIL_DirectModeConfig_t::dinLoc
```

DeprecatedNo longer used (ignored).

Definition at line 5528 of file common/rail_types.h

RAIL_VerifyConfig_t

The configuration array provided to RAIL for use by the radio state verification feature.

This structure will be populated with appropriate values by calling [RAIL_ConfigVerification\(\)](#). The application should not set or alter any of these structure elements.

Public Attributes

RAIL_Handle_t	correspondingHandle Internal verification tracking information.
uint32_t	nextIndexToVerify Internal verification tracking information.
RAIL_RadioConfig_t	override Internal verification tracking information.
RAIL_VerifyCallbackPtr_t	cb Internal verification tracking information.

Public Attribute Documentation

correspondingHandle

RAIL_Handle_t RAIL_VerifyConfig_t::correspondingHandle

Internal verification tracking information.

Definition at line 5610 of file common/rail_types.h

nextIndexToVerify

uint32_t RAIL_VerifyConfig_t::nextIndexToVerify

Internal verification tracking information.

Definition at line 5612 of file common/rail_types.h

override

RAIL_RadioConfig_t RAIL_VerifyConfig_t::override

Internal verification tracking information.

Definition at line 5614 of file common/rail_types.h

cb

RAIL_VerifyCallbackPtr_t RAIL_VerifyConfig_t::cb

Internal verification tracking information.

Definition at line 5616 of file common/rail_types.h

Events

Events

APIs related to events.

Enumerations

```
enum RAIL_Events_t {
    RAIL_EVENT_RSSI_AVERAGE_DONE_SHIFT = 0
    RAIL_EVENT_RX_ACK_TIMEOUT_SHIFT = 1
    RAIL_EVENT_RX_FIFO_ALMOST_FULL_SHIFT = 2
    RAIL_EVENT_RX_PACKET_RECEIVED_SHIFT = 3
    RAIL_EVENT_RX_PREAMBLE_LOST_SHIFT = 4
    RAIL_EVENT_RX_PREAMBLE_DETECT_SHIFT = 5
    RAIL_EVENT_RX_SYNC1_DETECT_SHIFT = 6
    RAIL_EVENT_RX_SYNC2_DETECT_SHIFT = 7
    RAIL_EVENT_RX_FRAME_ERROR_SHIFT = 8
    RAIL_EVENT_RX_FIFO_FULL_SHIFT = 9
    RAIL_EVENT_RX_FIFO_OVERFLOW_SHIFT = 10
    RAIL_EVENT_RX_ADDRESS_FILTERED_SHIFT = 11
    RAIL_EVENT_RX_TIMEOUT_SHIFT = 12
    RAIL_EVENT_RX_SCHEDULED_RX_STARTED_SHIFT = 13
    RAIL_EVENT_RX_SCHEDULED_RX_END_SHIFT = 14
    RAIL_EVENT_RX_SCHEDULED_RX_MISSED_SHIFT = 15
    RAIL_EVENT_RX_PACKET_ABORTED_SHIFT = 16
    RAIL_EVENT_RX_FILTER_PASSED_SHIFT = 17
    RAIL_EVENT_RX_TIMING_LOST_SHIFT = 18
    RAIL_EVENT_RX_TIMING_DETECT_SHIFT = 19
    RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE_SHIFT = 20
    RAIL_EVENT_RX_DUTY_CYCLE_RX_END_SHIFT = RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE_SHIFT
    RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND_SHIFT = 21
    RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND_SHIFT =
    RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND_SHIFT
    RAIL_EVENT_MFM_TX_BUFFER_DONE_SHIFT =
    RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND_SHIFT
    RAIL_EVENT_ZWAVE_BEAM_SHIFT = 22
    RAIL_EVENT_TX_FIFO_ALMOST_EMPTY_SHIFT = 23
    RAIL_EVENT_TX_PACKET_SENT_SHIFT = 24
    RAIL_EVENT_TXACK_PACKET_SENT_SHIFT = 25
    RAIL_EVENT_TX_ABORTED_SHIFT = 26
    RAIL_EVENT_TXACK_ABORTED_SHIFT = 27
    RAIL_EVENT_TX_BLOCKED_SHIFT = 28
    RAIL_EVENT_TXACK_BLOCKED_SHIFT = 29
    RAIL_EVENT_TX_UNDERFLOW_SHIFT = 30
    RAIL_EVENT_TXACK_UNDERFLOW_SHIFT = 31
    RAIL_EVENT_TX_CHANNEL_CLEAR_SHIFT = 32
    RAIL_EVENT_TX_CHANNEL_BUSY_SHIFT = 33
    RAIL_EVENT_TX_CCA_RETRY_SHIFT = 34
    RAIL_EVENT_TX_START_CCA_SHIFT = 35
    RAIL_EVENT_TX_STARTED_SHIFT = 36
    RAIL_EVENT_SCHEDULED_TX_STARTED_SHIFT = RAIL_EVENT_SCHEDULED_RX_STARTED_SHIFT
    RAIL_EVENT_TX_SCHEDULED_TX_MISSED_SHIFT = 37
    RAIL_EVENT_CONFIG_UNCHEDULED_SHIFT = 38
    RAIL_EVENT_CONFIG_SCHEDULED_SHIFT = 39
    RAIL_EVENT_SCHEDULER_STATUS_SHIFT = 40
    RAIL_EVENT_CAL_NEEDED_SHIFT = 41
    RAIL_EVENT_RF_SENSED_SHIFT = 42
    RAIL_EVENT_PA_PROTECTION_SHIFT = 43
    RAIL_EVENT_SIGNAL_DETECTED_SHIFT = 44
    RAIL_EVENT_IEEE802154_MODESWITCH_START_SHIFT = 45
    RAIL_EVENT_IEEE802154_MODESWITCH_END_SHIFT = 46
    RAIL_EVENT_DETECT_RSSI_THRESHOLD_SHIFT = 47
    RAIL_EVENT_THERMISTOR_DONE_SHIFT = 48
    RAIL_EVENT_TX_BLOCKED_TOO_HOT_SHIFT = 49
    RAIL_EVENT_TEMPERATURE_TOO_HOT_SHIFT = 50
    RAIL_EVENT_TEMPERATURE_COOL_DOWN_SHIFT = 51
    RAIL_EVENT_USER_MBOX_SHIFT = 52
}
```

RAIL events passed to the event callback.

Functions

[RAIL_Status_t](#) [RAIL_ConfigEvents](#)(RAIL_Handle_t railHandle, RAIL_Events_t mask, RAIL_Events_t events)
Configure radio events.

Macros

#define	RAIL_EVENTS_NONE 0ULL A value representing no events.
#define	RAIL_EVENT_RSSI_AVERAGE_DONE (1ULL << RAIL_EVENT_RSSI_AVERAGE_DONE_SHIFT) Occurs when the hardware-averaged RSSI is done in response to RAIL_StartAverageRssi() to indicate that the hardware has completed averaging.
#define	RAIL_EVENT_RX_ACK_TIMEOUT (1ULL << RAIL_EVENT_RX_ACK_TIMEOUT_SHIFT) Occurs when the Ack timeout expires while waiting to receive the sync word of an expected Ack.
#define	RAIL_EVENT_RX_FIFO_ALMOST_FULL (1ULL << RAIL_EVENT_RX_FIFO_ALMOST_FULL_SHIFT) Keeps occurring as long as the number of bytes in the receive FIFO exceeds the configured threshold value.
#define	RAIL_EVENT_RX_PACKET_RECEIVED (1ULL << RAIL_EVENT_RX_PACKET_RECEIVED_SHIFT) Occurs whenever a packet is received with RAIL_RX_PACKET_READY_SUCCESS or RAIL_RX_PACKET_READY_CRC_ERROR .
#define	RAIL_EVENT_RX_PREAMBLE_LOST (1ULL << RAIL_EVENT_RX_PREAMBLE_LOST_SHIFT) Occurs when the radio has lost a preamble.
#define	RAIL_EVENT_RX_PREAMBLE_DETECT (1ULL << RAIL_EVENT_RX_PREAMBLE_DETECT_SHIFT) Occurs when the radio has detected a preamble.
#define	RAIL_EVENT_RX_SYNC1_DETECT (1ULL << RAIL_EVENT_RX_SYNC1_DETECT_SHIFT) Occurs when the first sync word is detected.
#define	RAIL_EVENT_RX_SYNC2_DETECT (1ULL << RAIL_EVENT_RX_SYNC2_DETECT_SHIFT) Occurs when the second sync word is detected.
#define	RAIL_EVENT_RX_FRAME_ERROR (1ULL << RAIL_EVENT_RX_FRAME_ERROR_SHIFT) Occurs when a receive is aborted with RAIL_RX_PACKET_ABORT_CRC_ERROR which only happens after any filtering has passed.
#define	RAIL_EVENT_RX_FIFO_FULL (1ULL << RAIL_EVENT_RX_FIFO_FULL_SHIFT) When using RAIL_RxDataSource_t::RX_PACKET_DATA this event occurs coincident to a receive packet completion event in which the receive FIFO or any supplemental packet metadata FIFO (see Data Management) are full and further packet reception is jeopardized.
#define	RAIL_EVENT_RX_FIFO_OVERFLOW (1ULL << RAIL_EVENT_RX_FIFO_OVERFLOW_SHIFT) When using RAIL_RxDataSource_t::RX_PACKET_DATA this event occurs when a receive is aborted with RAIL_RX_PACKET_ABORT_OVERFLOW due to overflowing the receive FIFO or any supplemental packet metadata FIFO (see Data Management).
#define	RAIL_EVENT_RX_ADDRESS_FILTERED (1ULL << RAIL_EVENT_RX_ADDRESS_FILTERED_SHIFT) Occurs when a receive is aborted with RAIL_RX_PACKET_ABORT_FILTERED because its address does not match the filtering settings.
#define	RAIL_EVENT_RX_TIMEOUT (1ULL << RAIL_EVENT_RX_TIMEOUT_SHIFT) Occurs when an RX event times out.
#define	RAIL_EVENT_SCHEDULED_RX_STARTED (1ULL << RAIL_EVENT_SCHEDULED_RX_STARTED_SHIFT) Occurs when a scheduled RX begins turning on the receiver.
#define	RAIL_EVENT_RX_SCHEDULED_RX_END (1ULL << RAIL_EVENT_RX_SCHEDULED_RX_END_SHIFT) Occurs when the scheduled RX window ends.
#define	RAIL_EVENT_RX_SCHEDULED_RX_MISSED (1ULL << RAIL_EVENT_RX_SCHEDULED_RX_MISSED_SHIFT) Occurs when start of a scheduled receive is missed.

#define	RAIL_EVENT_RX_PACKET_ABORTED (1ULL << RAIL_EVENT_RX_PACKET_ABORTED_SHIFT)
	Occurs when a receive is aborted during filtering with RAIL_RX_PACKET_ABORT_FORMAT or after filtering with RAIL_RX_PACKET_ABORT_ABORTED for reasons other than address filtering mismatch (which triggers RAIL_EVENT_RX_ADDRESS_FILTERED instead).
#define	RAIL_EVENT_RX_FILTER_PASSED (1ULL << RAIL_EVENT_RX_FILTER_PASSED_SHIFT)
	Occurs when the packet has passed any configured address and frame filtering options.
#define	RAIL_EVENT_RX_TIMING_LOST (1ULL << RAIL_EVENT_RX_TIMING_LOST_SHIFT)
	Occurs when the modem timing is lost.
#define	RAIL_EVENT_RX_TIMING_DETECT (1ULL << RAIL_EVENT_RX_TIMING_DETECT_SHIFT)
	Occurs when the modem timing is detected.
#define	RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE (1ULL << RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE_SHIFT)
	Occurs when RX Channel Hopping is enabled and channel hopping finishes receiving on the last channel in its sequence.
#define	RAIL_EVENT_RX_DUTY_CYCLE_RX_END (1ULL << RAIL_EVENT_RX_DUTY_CYCLE_RX_END_SHIFT)
	Occurs during RX duty cycle mode when the radio finishes its time in receive mode.
#define	RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND (1ULL << RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND_SHIFT)
	Indicate a Data Request is received when using IEEE 802.15.4 functionality.
#define	RAIL_EVENT_ZWAVE_BEAM (1ULL << RAIL_EVENT_ZWAVE_BEAM_SHIFT)
	Indicate a Z-Wave Beam Request relevant to the node was received.
#define	RAIL_EVENT_MFM_TX_BUFFER_DONE (1ULL << RAIL_EVENT_MFM_TX_BUFFER_DONE_SHIFT)
	Indicate a MFM buffer has completely transmitted.
#define	RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND (1ULL << RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND_SHIFT)
	Indicate a request for populating Z-Wave LR Ack packet.
#define	RAIL_EVENTS_RX_COMPLETION undefined
	The mask representing all events that determine the end of a received packet.
#define	RAIL_EVENT_TX_FIFO_ALMOST_EMPTY (1ULL << RAIL_EVENT_TX_FIFO_ALMOST_EMPTY_SHIFT)
	Occurs when the number of bytes in the transmit FIFO falls below the configured threshold value.
#define	RAIL_EVENT_TX_PACKET_SENT (1ULL << RAIL_EVENT_TX_PACKET_SENT_SHIFT)
	Occurs after a packet has been transmitted.
#define	RAIL_EVENT_TXACK_PACKET_SENT (1ULL << RAIL_EVENT_TXACK_PACKET_SENT_SHIFT)
	Occurs after an Ack packet has been transmitted.
#define	RAIL_EVENT_TX_ABORTED (1ULL << RAIL_EVENT_TX_ABORTED_SHIFT)
	Occurs when a transmit is aborted by the user.
#define	RAIL_EVENT_TXACK_ABORTED (1ULL << RAIL_EVENT_TXACK_ABORTED_SHIFT)
	Occurs when an Ack transmit is aborted by the user.
#define	RAIL_EVENT_TX_BLOCKED (1ULL << RAIL_EVENT_TX_BLOCKED_SHIFT)
	Occurs when a transmit is blocked from occurring because RAIL_EnableTxHoldOff() was called.
#define	RAIL_EVENT_TXACK_BLOCKED (1ULL << RAIL_EVENT_TXACK_BLOCKED_SHIFT)
	Occurs when an Ack transmit is blocked from occurring because RAIL_EnableTxHoldOff() was called.
#define	RAIL_EVENT_TX_UNDERFLOW (1ULL << RAIL_EVENT_TX_UNDERFLOW_SHIFT)
	Occurs when the transmit buffer underflows.

#define	RAIL_EVENT_TXACK_UNDERFLOW (1ULL << RAIL_EVENT_TXACK_UNDERFLOW_SHIFT) Occurs when the Ack transmit buffer underflows.
#define	RAIL_EVENT_TX_CHANNEL_CLEAR (1ULL << RAIL_EVENT_TX_CHANNEL_CLEAR_SHIFT) Occurs when Carrier Sense Multiple Access (CSMA) or Listen Before Talk (LBT) succeeds.
#define	RAIL_EVENT_TX_CHANNEL_BUSY (1ULL << RAIL_EVENT_TX_CHANNEL_BUSY_SHIFT) Occurs when Carrier Sense Multiple Access (CSMA) or Listen Before Talk (LBT) fails.
#define	RAIL_EVENT_TX_CCA_RETRY (1ULL << RAIL_EVENT_TX_CCA_RETRY_SHIFT) Occurs during CSMA or LBT when an individual Clear Channel Assessment (CCA) check fails, but there are more tries needed before the overall operation completes.
#define	RAIL_EVENT_TX_START_CCA (1ULL << RAIL_EVENT_TX_START_CCA_SHIFT) Occurs when the receiver is activated to perform a Clear Channel Assessment (CCA) check.
#define	RAIL_EVENT_TX_STARTED (1ULL << RAIL_EVENT_TX_STARTED_SHIFT) Occurs when the radio starts transmitting a normal packet on the air.
#define	RAIL_TX_STARTED_BYTES 0U A value to pass as RAIL_GetTxTimePreambleStart() totalPacketBytes parameter to retrieve the RAIL_EVENT_TX_STARTED timestamp.
#define	RAIL_EVENT_SCHEDULED_TX_STARTED (1ULL << RAIL_EVENT_SCHEDULED_TX_STARTED_SHIFT) Occurs when a scheduled TX begins turning on the transmitter.
#define	RAIL_EVENT_TX_SCHEDULED_TX_MISSED (1ULL << RAIL_EVENT_TX_SCHEDULED_TX_MISSED_SHIFT) Occurs when the start of a scheduled transmit is missed.
#define	RAIL_EVENTS_TX_COMPLETION undefined A mask representing all events that determine the end of a transmitted packet.
#define	RAIL_EVENTS_TXACK_COMPLETION undefined A mask representing all events that determine the end of a transmitted Ack packet.
#define	RAIL_EVENT_CONFIG_UNCHEDULED (1ULL << RAIL_EVENT_CONFIG_UNCHEDULED_SHIFT) Occurs when the scheduler switches away from this configuration.
#define	RAIL_EVENT_CONFIG_SCHEDULED (1ULL << RAIL_EVENT_CONFIG_SCHEDULED_SHIFT) Occurs when the scheduler switches to this configuration.
#define	RAIL_EVENT_SCHEDULER_STATUS (1ULL << RAIL_EVENT_SCHEDULER_STATUS_SHIFT) Occurs when the scheduler has a status to report.
#define	RAIL_EVENT_CAL_NEEDED (1ULL << RAIL_EVENT_CAL_NEEDED_SHIFT) Occurs when the application needs to run a calibration, as determined by the RAIL library.
#define	RAIL_EVENT_RF_SENSED (1ULL << RAIL_EVENT_RF_SENSED_SHIFT) Occurs when RF energy is sensed from the radio.
#define	RAIL_EVENT_PA_PROTECTION (1ULL << RAIL_EVENT_PA_PROTECTION_SHIFT) Occurs when PA protection circuit kicks in.
#define	RAIL_EVENT_SIGNAL_DETECTED (1ULL << RAIL_EVENT_SIGNAL_DETECTED_SHIFT) Occurs after enabling the signal detection using RAIL_BLE_EnableSignalDetection() or RAIL_IEEE802154_EnableSignalDetection() when a signal is detected.
#define	RAIL_EVENT_IEEE802154_MODESWITCH_START (1ULL << RAIL_EVENT_IEEE802154_MODESWITCH_START_SHIFT) Occurs when a Wi-SUN mode switch packet has been received, after switching to the new PHY.

- #define RAIL_EVENT_IEEE802154_MODESWITCH_END (1ULL << RAIL_EVENT_IEEE802154_MODESWITCH_END_SHIFT)
Occurs when switching back to the original base PHY in effect prior to the Wi-SUN mode switch reception.
- #define RAIL_EVENT_DETECT_RSSI_THRESHOLD (1ULL << RAIL_EVENT_DETECT_RSSI_THRESHOLD_SHIFT)
Occurs when the sampled RSSI is above the threshold set by [RAIL_SetRssiDetectThreshold\(\)](#).
- #define RAIL_EVENT_THERMISTOR_DONE (1ULL << RAIL_EVENT_THERMISTOR_DONE_SHIFT)
Occurs when the thermistor has finished its measurement in response to [RAIL_StartThermistorMeasurement\(\)](#).
- #define RAIL_EVENT_TX_BLOCKED_TOO_HOT (1ULL << RAIL_EVENT_TX_BLOCKED_TOO_HOT_SHIFT)
Occurs when a Tx has been blocked because of temperature exceeding the safety threshold.
- #define RAIL_EVENT_TEMPERATURE_TOO_HOT (1ULL << RAIL_EVENT_TEMPERATURE_TOO_HOT_SHIFT)
Occurs when die internal temperature exceeds the temperature threshold subtracted by the cool down parameter from [RAIL_ChipTempConfig_t](#).
- #define RAIL_EVENT_TEMPERATURE_COOL_DOWN (1ULL << RAIL_EVENT_TEMPERATURE_COOL_DOWN_SHIFT)
Occurs when die internal temperature falls below the temperature threshold subtracted by the cool down parameter from [RAIL_ChipTempConfig_t](#).
- #define RAIL_EVENT_USER_MBOX (1ULL << RAIL_EVENT_USER_MBOX_SHIFT)
Occurs when the user received a mailbox message.
- #define RAIL_EVENTS_ALL 0xFFFFFFFFFFFFFFFFULL
A value representing all possible events.

Enumeration Documentation

RAIL_Events_t

RAIL_Events_t

RAIL events passed to the event callback.

More than one event may be indicated due to interrupt latency.

Enumerator	
RAIL_EVENT_RSSI_AVERAGE_DONE_SHIFT	Shift position of RAIL_EVENT_RSSI_AVERAGE_DONE bit.
RAIL_EVENT_RX_ACK_TIMEOUT_SHIFT	Shift position of RAIL_EVENT_RX_ACK_TIMEOUT bit.
RAIL_EVENT_RX_FIFO_ALMOST_FULL_SHIFT	Shift position of RAIL_EVENT_RX_FIFO_ALMOST_FULL bit.
RAIL_EVENT_RX_PACKET_RECEIVED_SHIFT	Shift position of RAIL_EVENT_RX_PACKET_RECEIVED bit.
RAIL_EVENT_RX_PREAMBLE_LOST_SHIFT	Shift position of RAIL_EVENT_RX_PREAMBLE_LOST bit.
RAIL_EVENT_RX_PREAMBLE_DETECT_SHIFT	Shift position of RAIL_EVENT_RX_PREAMBLE_DETECT bit.
RAIL_EVENT_RX_SYNC1_DETECT_SHIFT	Shift position of RAIL_EVENT_RX_SYNC1_DETECT bit.
RAIL_EVENT_RX_SYNC2_DETECT_SHIFT	Shift position of RAIL_EVENT_RX_SYNC2_DETECT bit.
RAIL_EVENT_RX_FRAME_ERROR_SHIFT	Shift position of RAIL_EVENT_RX_FRAME_ERROR bit.
RAIL_EVENT_RX_FIFO_FULL_SHIFT	Shift position of RAIL_EVENT_RX_FIFO_FULL bit.
RAIL_EVENT_RX_FIFO_OVERFLOW_SHIFT	Shift position of RAIL_EVENT_RX_FIFO_OVERFLOW bit.
RAIL_EVENT_RX_ADDRESS_FILTERED_SHIFT	Shift position of RAIL_EVENT_RX_ADDRESS_FILTERED bit.
RAIL_EVENT_RX_TIMEOUT_SHIFT	Shift position of RAIL_EVENT_RX_TIMEOUT bit.
RAIL_EVENT_SCHEDULED_RX_STARTED_SHIFT	Shift position of RAIL_EVENT_SCHEDULED_RX_STARTED bit.
RAIL_EVENT_RX_SCHEDULED_RX_END_SHIFT	Shift position of RAIL_EVENT_RX_SCHEDULED_RX_END bit.

RAIL_EVENT_RX_SCHEDULED_RX_MISSED_SHIFT	Shift position of RAIL_EVENT_RX_SCHEDULED_RX_MISSED bit.
RAIL_EVENT_RX_PACKET_ABORTED_SHIFT	Shift position of RAIL_EVENT_RX_PACKET_ABORTED bit.
RAIL_EVENT_RX_FILTER_PASSED_SHIFT	Shift position of RAIL_EVENT_RX_FILTER_PASSED bit.
RAIL_EVENT_RX_TIMING_LOST_SHIFT	Shift position of RAIL_EVENT_RX_TIMING_LOST bit.
RAIL_EVENT_RX_TIMING_DETECT_SHIFT	Shift position of RAIL_EVENT_RX_TIMING_DETECT bit.
RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE_SHIFT	Shift position of RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE bit.
RAIL_EVENT_RX_DUTY_CYCLE_RX_END_SHIFT	Shift position of RAIL_EVENT_RX_DUTY_CYCLE_RX_END bit.
RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND_SHIFT	Shift position of RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND bit.
RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND_SHIFT	Shift position of RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND_SHIFT bit.
RAIL_EVENT_MFM_TX_BUFFER_DONE_SHIFT	Shift position of RAIL_EVENT_MFM_TX_BUFFER_DONE bit.
RAIL_EVENT_ZWAVE_BEAM_SHIFT	Shift position of RAIL_EVENT_ZWAVE_BEAM bit.
RAIL_EVENT_TX_FIFO_ALMOST_EMPTY_SHIFT	Shift position of RAIL_EVENT_TX_FIFO_ALMOST_EMPTY bit.
RAIL_EVENT_TX_PACKET_SENT_SHIFT	Shift position of RAIL_EVENT_TX_PACKET_SENT bit.
RAIL_EVENT_TXACK_PACKET_SENT_SHIFT	Shift position of RAIL_EVENT_TXACK_PACKET_SENT bit.
RAIL_EVENT_TX_ABORTED_SHIFT	Shift position of RAIL_EVENT_TX_ABORTED bit.
RAIL_EVENT_TXACK_ABORTED_SHIFT	Shift position of RAIL_EVENT_TXACK_ABORTED bit.
RAIL_EVENT_TX_BLOCKED_SHIFT	Shift position of RAIL_EVENT_TX_BLOCKED bit.
RAIL_EVENT_TXACK_BLOCKED_SHIFT	Shift position of RAIL_EVENT_TXACK_BLOCKED bit.
RAIL_EVENT_TX_UNDERFLOW_SHIFT	Shift position of RAIL_EVENT_TX_UNDERFLOW bit.
RAIL_EVENT_TXACK_UNDERFLOW_SHIFT	Shift position of RAIL_EVENT_TXACK_UNDERFLOW bit.
RAIL_EVENT_TX_CHANNEL_CLEAR_SHIFT	Shift position of RAIL_EVENT_TX_CHANNEL_CLEAR bit.
RAIL_EVENT_TX_CHANNEL_BUSY_SHIFT	Shift position of RAIL_EVENT_TX_CHANNEL_BUSY bit.
RAIL_EVENT_TX_CCA_RETRY_SHIFT	Shift position of RAIL_EVENT_TX_CCA_RETRY bit.
RAIL_EVENT_TX_START_CCA_SHIFT	Shift position of RAIL_EVENT_TX_START_CCA bit.
RAIL_EVENT_TX_STARTED_SHIFT	Shift position of RAIL_EVENT_TX_STARTED bit.
RAIL_EVENT_SCHEDULED_TX_STARTED_SHIFT	Shift position of RAIL_EVENT_SCHEDULED_TX_STARTED bit.
RAIL_EVENT_TX_SCHEDULED_TX_MISSED_SHIFT	Shift position of RAIL_EVENT_TX_SCHEDULED_TX_MISSED bit.
RAIL_EVENT_CONFIG_UNCHEDULED_SHIFT	Shift position of RAIL_EVENT_CONFIG_UNCHEDULED bit.
RAIL_EVENT_CONFIG_SCHEDULED_SHIFT	Shift position of RAIL_EVENT_CONFIG_SCHEDULED bit.
RAIL_EVENT_SCHEDULER_STATUS_SHIFT	Shift position of RAIL_EVENT_SCHEDULER_STATUS bit.
RAIL_EVENT_CAL_NEEDED_SHIFT	Shift position of RAIL_EVENT_CAL_NEEDED bit.
RAIL_EVENT_RF_SENSED_SHIFT	Shift position of RAIL_EVENT_RF_SENSED bit.
RAIL_EVENT_PA_PROTECTION_SHIFT	Shift position of RAIL_EVENT_PA_PROTECTION bit.
RAIL_EVENT_SIGNAL_DETECTED_SHIFT	Shift position of RAIL_EVENT_SIGNAL_DETECTED bit.
RAIL_EVENT_IEEE802154_MODESWITCH_START_SHIFT	Shift position of RAIL_EVENT_IEEE802154_MODESWITCH_START bit.
RAIL_EVENT_IEEE802154_MODESWITCH_END_SHIFT	Shift position of RAIL_EVENT_IEEE802154_MODESWITCH_END bit.

RAIL_EVENT_DETECT_RSSI_THRESHOLD_SHIFT	Shift position of RAIL_EVENT_DETECT_RSSI_THRESHOLD bit.
RAIL_EVENT_THERMISTOR_DONE_SHIFT	Shift position of RAIL_EVENT_THERMISTOR_DONE bit.
RAIL_EVENT_TX_BLOCKED_TOO_HOT_SHIFT	Shift position of RAIL_EVENT_TX_BLOCKED_TOO_HOT bit.
RAIL_EVENT_TEMPERATURE_TOO_HOT_SHIFT	Shift position of RAIL_EVENT_TEMPERATURE_TOO_HOT bit.
RAIL_EVENT_TEMPERATURE_COOL_DOWN_SHIFT	Shift position of RAIL_EVENT_TEMPERATURE_COOL_DOWN bit.
RAIL_EVENT_USER_MBOX_SHIFT	Shift position of RAIL_EVENT_USER_MBOX bit.

Definition at line 807 of file common/rail_types.h

Function Documentation

RAIL_ConfigEvents

RAIL_Status_t RAIL_ConfigEvents ([RAIL_Handle_t](#) railHandle, [RAIL_Events_t](#) mask, [RAIL_Events_t](#) events)

Configure radio events.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mask	A bitmask of events to configure.
[in]	events	A bitmask of events to trigger RAIL_Config_t::eventsCallback .

Returns

- Status code indicating success of the function call.
- Sets up which radio interrupts generate a RAIL event. The full list of events is in [RAIL_Events_t](#).

Definition at line 1688 of file common/rail.h

External Thermistor

External Thermistor

APIs to measure temperature using an external thermistor.

This feature allows reading the temperature via an external thermistor on chips that support it. This will require connecting the necessary components and configuring the pins as required.

Modules

- [RAIL_HFXOThermistorConfig_t](#)
- [RAIL_HFXOCompensationConfig_t](#)

Functions

RAIL_Status_t	RAIL_StartThermistorMeasurement (RAIL_Handle_t railHandle) Start a thermistor measurement.
RAIL_Status_t	RAIL_GetThermistorImpedance (RAIL_Handle_t railHandle, uint32_t *thermistorImpedance) Get the thermistor impedance measurement and return RAIL_INVALID_THERMISTOR_VALUE if the thermistor is not properly configured or the thermistor measurement is not ready.
RAIL_Status_t	RAIL_ConvertThermistorImpedance (RAIL_Handle_t railHandle, uint32_t thermistorImpedance, int16_t *thermistorTemperatureC) Convert the thermistor impedance into temperature, in Celsius.
RAIL_Status_t	RAIL_ComputeHFXOPPMError (RAIL_Handle_t railHandle, int16_t crystalTemperatureC, int8_t *crystalPPMError) Compute the crystal PPM deviation from the thermistor temperature.
RAIL_Status_t	RAIL_ConfigHFXOThermistor (RAIL_Handle_t railHandle, const RAIL_HFXOThermistorConfig_t *pHfxoThermistorConfig) Configure the GPIO for thermistor usage.
RAIL_Status_t	RAIL_ConfigHFXOCompensation (RAIL_Handle_t railHandle, const RAIL_HFXOCompensationConfig_t *pHfxoCompensationConfig) Configure the temperature parameters for HFXO compensation.
RAIL_Status_t	RAIL_GetHFXOCompensationConfig (RAIL_Handle_t railHandle, RAIL_HFXOCompensationConfig_t *pHfxoCompensationConfig) Get the temperature parameters for HFXO compensation.
RAIL_Status_t	RAIL_CompensateHFXO (RAIL_Handle_t railHandle, int8_t crystalPPMError) Compute a frequency offset and compensate HFXO accordingly.

Macros

#define	RAIL_INVALID_THERMISTOR_VALUE (0xFFFFFFFFU) A sentinel value to indicate an invalid thermistor measurement value.
#define	RAIL_INVALID_PPM_VALUE (-128) A sentinel value to indicate an invalid PPM calculation value.

Function Documentation

RAIL_StartThermistorMeasurement

RAIL_Status_t RAIL_StartThermistorMeasurement ([RAIL_Handle_t](#) railHandle)

Start a thermistor measurement.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- Status code indicating success of the function call. Returns [RAIL_STATUS_INVALID_STATE](#) if the thermistor is started while the radio is transmitting.

To get the thermistor impedance, call the function [RAIL_GetThermistorImpedance\(\)](#). On platforms having [RAIL_SUPPORTS_EXTERNAL_THERMISTOR](#), this function reconfigures GPIO_THMSW_EN_PIN located in GPIO_THMSW_EN_PORT. To locate this pin, refer to the data sheet or appropriate header files of the device. For proper operation, [RAIL_Init\(\)](#) must be called before using this function.

Note

- When an EFF is attached, this function must not be called during transmit.

Warnings

- This API is not safe to use in a multiprotocol app.
- As this function relies on EMU, GPIO and HFXO access and RAIL is meant to run in TrustZone non-secure world, it is not supported if EMU, GPIO or HFXO are configured as secure peripheral and it will return [RAIL_STATUS_INVALID_CALL](#).

Definition at line 6731 of file common/rail.h

RAIL_GetThermistorImpedance

RAIL_Status_t RAIL_GetThermistorImpedance ([RAIL_Handle_t](#) railHandle, uint32_t * thermistorImpedance)

Get the thermistor impedance measurement and return [RAIL_INVALID_THERMISTOR_VALUE](#) if the thermistor is not properly configured or the thermistor measurement is not ready.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.
[out]	thermistorImpedance	A pointer to a uint32_t updated with the current thermistor impedance measurement in Ohms.

Returns

- Status code indicating success of the function call.

Note

- This function is already called in [RAIL_CalibrateHFXO\(\)](#). It does not need to be manually called during the compensation sequence.

Definition at line 6746 of file common/rail.h

RAIL_ConvertThermistorImpedance

```
RAIL_Status_t RAIL_ConvertThermistorImpedance (RAIL_Handle_t railHandle, uint32_t thermistorImpedance, int16_t * thermistorTemperatureC)
```

Convert the thermistor impedance into temperature, in Celsius.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.
[in]	thermistorImpedance	Current thermistor impedance measurement in Ohms.
[out]	thermistorTemperatureC	A non-NULL pointer to an int16_t updated with the current thermistor temperature in eighth of Celsius degrees.

Returns

- Status code indicating success of the function call.

A version of this function is provided in the [Thermistor Utility](#) plugin for Silicon Labs radio boards. For custom boards this function can be modified and re-implemented as needed in the plugin.

Note

- The [Thermistor Utility](#) plugin is mandatory on EFR32xG25 platforms.

Definition at line 6765 of file common/rail.h

RAIL_ComputeHFXOPPMError

```
RAIL_Status_t RAIL_ComputeHFXOPPMError (RAIL_Handle_t railHandle, int16_t crystalTemperatureC, int8_t * crystalPPMError)
```

Compute the crystal PPM deviation from the thermistor temperature.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.
[in]	crystalTemperatureC	Current crystal temperature, in Celsius.
[out]	crystalPPMError	A non-NULL pointer to an int8_t updated with the current ppm error in ppm units.

Returns

- Status code indicating success of the function call.

A version of this function is provided in the [Thermistor Utility](#) plugin for Silicon Labs radio boards. For custom boards this function can be modified and re-implemented as needed in the plugin.

Note

- The [Thermistor Utility](#) plugin is mandatory on EFR32xG25 platforms.

Definition at line 6784 of file common/rail.h

RAIL_ConfigHFXOThermistor

```
RAIL_Status_t RAIL_ConfigHFXOThermistor (RAIL_Handle_t railHandle, const RAIL_HFXOThermistorConfig_t * pHfxoThermistorConfig)
```

Configure the GPIO for thermistor usage.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.
[in]	pHfxoThermistorConfig	A non-NULL pointer to the thermistor configuration indicating the GPIO port and pin to use.

Returns

- Status code indicating the result of the function call.

Note

- The port and pin that must be passed in [RAIL_HFXOThermistorConfig_t](#) are GPIO_THMSW_EN_PORT and GPIO_THMSW_EN_PIN respectively.

Warnings

- As this function relies on GPIO access and RAIL is meant to run in TrustZone non-secure world, it is not supported if GPIO is configured as secure peripheral and it will return [RAIL_STATUS_INVALID_CALL](#).

Definition at line 6803 of file common/rail.h

RAIL_ConfigHFXOCompensation

```
RAIL_Status_t RAIL_ConfigHFXOCompensation (RAIL\_Handle\_t railHandle, const RAIL\_HFXOCompensationConfig\_t *pHfxoCompensationConfig)
```

Configure the temperature parameters for HFXO compensation.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	pHfxoCompensationConfig	A non-NULL pointer to HFXO compensation parameters indicating the temperature variations used to trigger a compensation.

Returns

- Status code indicating the result of the function call.

Note

- This function must be called after [RAIL_ConfigHFXOThermistor\(\)](#) to succeed.

In [RAIL_HFXOCompensationConfig_t](#), deltaNominal and deltaCritical define the temperature variation triggering a new compensation. The field zoneTemperatureC defines the temperature separating the nominal case (below) from the critical one (above).

When enabled and either deltaNominal or deltaCritical are exceeded, RAIL raises event [RAIL_EVENT_CAL_NEEDED](#) with [RAIL_CAL_TEMP_HFXO](#) bit set. The API [RAIL_StartThermistorMeasurement\(\)](#) must be called afterwards. The latter will raise [RAIL_EVENT_THERMISTOR_DONE](#) with calibration bit [RAIL_CAL_COMPENSATE_HFXO](#) set and a call to [RAIL_CalibrateHFXO\(\)](#) must follow.

Note

- Set deltaNominal and deltaCritical to 0 to perform compensation after each transmit.

Definition at line 6832 of file common/rail.h

RAIL_GetHFXOCompensationConfig

```
RAIL_Status_t RAIL_GetHFXOCCompensationConfig (RAIL_Handle_t railHandle, RAIL_HFXOCCompensationConfig_t * pHfxoCompensationConfig)
```

Get the temperature parameters for HFXO compensation.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	pHfxoCompensationConfig	A non-NULL pointer to HFXO compensation parameters filled in by the function.

Returns

- Status code indicating the result of the function call.

Definition at line 6843 of file common/rail.h

RAIL_CompensateHFXO

```
RAIL_Status_t RAIL_CompensateHFXO (RAIL_Handle_t railHandle, int8_t crystalPPMError)
```

Compute a frequency offset and compensate HFXO accordingly.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	crystalPPMError	The current ppm error. Positive values indicate the HFXO frequency is too high; negative values indicate it's too low.

Returns

- Status code indicating success of the function call.

Note

- This function only works for platforms having RAIL_SUPPORTS_EXTERNAL_THERMISTOR alongside RAIL_SUPPORTS_HFXO_COMPENSATION.

Definition at line 6857 of file common/rail.h

RAIL_HFXOThermistorConfig_t

Configure the port and pin of the thermistor.

Note

- This configuration is chip OPN dependent.

Public Attributes

uint8_t	port	The GPIO port to access the thermistor.
uint8_t	pin	The GPIO pin to set the thermistor.

Public Attribute Documentation

port

```
uint8_t RAIL_HFXOThermistorConfig_t::port
```

The GPIO port to access the thermistor.

Definition at line 4608 of file common/rail_types.h

pin

```
uint8_t RAIL_HFXOThermistorConfig_t::pin
```

The GPIO pin to set the thermistor.

Definition at line 4612 of file common/rail_types.h

RAIL_HFXOCompensationConfig_t

Set compensation specific parameters.

Public Attributes

bool	enableCompensation	Indicates whether the HFXO compensation in temperature is activated.
int8_t	zoneTemperatureC	The temperature reference delimiting the nominal zone from the critical one.
uint8_t	deltaNominal	The temperature shift used to start a new compensation, in the nominal zone.
uint8_t	deltaCritical	The temperature shift used to start a new compensation, in the critical zone.

Public Attribute Documentation

enableCompensation

```
bool RAIL_HFXOCompensationConfig_t::enableCompensation
```

Indicates whether the HFXO compensation in temperature is activated.

Definition at line 4623 of file common/rail_types.h

zoneTemperatureC

```
int8_t RAIL_HFXOCompensationConfig_t::zoneTemperatureC
```

The temperature reference delimiting the nominal zone from the critical one.

This field is relevant if enableCompensation is set to true.

Definition at line 4628 of file common/rail_types.h

deltaNominal

```
uint8_t RAIL_HFXOCompensationConfig_t::deltaNominal
```

The temperature shift used to start a new compensation, in the nominal zone.

This field is relevant if enableCompensation is set to true.

Definition at line 4633 of file common/rail_types.h

deltaCritical

```
uint8_t RAIL_HFXOCompensationConfig_t::deltaCritical
```

The temperature shift used to start a new compensation, in the critical zone.

This field is relevant if enableCompensation is set to true.

Definition at line 4638 of file common/rail_types.h

Features

Features

Overview of support for various features across hardware platforms.

These defines can be used at compile time to determine which features are available on your platform. However, keep in mind that these defines hold true for chip families. Your specific part may have further restrictions (band limitations, power amplifier restrictions, and so on) on top of those listed below, for which runtime `RAIL_*Supports*()` APIs can be used to check availability on a particular chip (after `RAIL_Init()` has been called). In general, an attempt to call an API that is not supported on your chip family as listed below will result in a `RAIL_STATUS_INVALID_CALL`.

Functions

bool	<code>RAIL_Supports2p4GHzBand(RAIL_Handle_t railHandle)</code> Indicate whether RAIL supports 2.4 GHz band operation on this chip.
bool	<code>RAIL_SupportsSubGHzBand(RAIL_Handle_t railHandle)</code> Indicate whether RAIL supports SubGHz band operation on this chip.
bool	<code>RAIL_SupportsDualBand(RAIL_Handle_t railHandle)</code> Indicate whether this chip supports dual 2.4 GHz and SubGHz band operation.
bool	<code>RAIL_SupportsAddrFilterAddressBitMask(RAIL_Handle_t railHandle)</code> Indicate whether this chip supports bit masked address filtering.
bool	<code>RAIL_SupportsAddrFilterMask(RAIL_Handle_t railHandle)</code> Indicate whether this chip supports address filter mask information for incoming packets in <code>RAIL_RxPacketInfo_t::filterMask</code> and <code>RAIL_IEEE802154_Address_t::filterMask</code> .
bool	<code>RAIL_SupportsAlternateTxPower(RAIL_Handle_t railHandle)</code> Indicate whether this chip supports alternate TX power settings.
bool	<code>RAIL_SupportsAntennaDiversity(RAIL_Handle_t railHandle)</code> Indicate whether this chip supports antenna diversity.
bool	<code>RAIL_SupportsPathDiversity(RAIL_Handle_t railHandle)</code> Indicate whether this chip supports internal RF path diversity.
bool	<code>RAIL_SupportsAuxAdc(RAIL_Handle_t railHandle)</code> Indicate whether RAIL supports AUXADC measurements on this chip.
bool	<code>RAIL_SupportsChannelHopping(RAIL_Handle_t railHandle)</code> Indicate whether RAIL supports channel hopping on this chip.
bool	<code>RAIL_SupportsDirectMode(RAIL_Handle_t railHandle)</code> Indicate whether this chip supports direct mode.
bool	<code>RAIL_SupportsDualSyncWords(RAIL_Handle_t railHandle)</code> Indicate whether this chip supports dual sync words.
bool	<code>RAIL_SupportsTxRepeatStartToStart(RAIL_Handle_t railHandle)</code> Indicate whether this chip supports start to start TX repeats.
bool	<code>RAIL_SupportsVdet(RAIL_Handle_t railHandle)</code> Indicate whether this chip supports VDET.

bool	RAIL_SupportsExternalThermistor (RAIL_Handle_t railHandle) Indicate whether RAIL supports thermistor measurements on this chip.
bool	RAIL_SupportsHFXOCompensation (RAIL_Handle_t railHandle) Indicate whether RAIL supports HFXO compensation on this chip.
bool	RAIL_SupportsMfm (RAIL_Handle_t railHandle) Indicate whether this chip supports MFM protocol.
bool	RAIL_SupportsOFDMPA (RAIL_Handle_t railHandle) Indicate whether RAIL supports OFDM band operation on this chip.
bool	RAIL_SupportsPrecisionLFRCO (RAIL_Handle_t railHandle) Indicate whether this chip supports a high-precision LFRCO.
bool	RAIL_SupportsRadioEntropy (RAIL_Handle_t railHandle) Indicate whether this chip supports radio entropy.
bool	RAIL_SupportsRfSenseEnergyDetection (RAIL_Handle_t railHandle) Indicate whether RAIL supports RF Sense Energy Detection Mode on this chip.
bool	RAIL_SupportsRfSenseSelectiveOok (RAIL_Handle_t railHandle) Indicate whether RAIL supports RF Sense Selective(OOK) Mode on this chip.
bool	RAIL_SupportsRssiDetectThreshold (RAIL_Handle_t railHandle) Indicate whether this chip supports configurable RSSI threshold set by RAIL_SetRssiDetectThreshold() .
bool	RAIL_SupportsRxDirectModeDataToFifo (RAIL_Handle_t railHandle) Indicate whether this chip supports RX direct mode data to FIFO.
bool	RAIL_SupportsRxRawData (RAIL_Handle_t railHandle) Indicate whether this chip supports raw RX data sources other than RAIL_RxDataSource_t::RX_PACKET_DATA .
bool	RAIL_SupportsSQPhy (RAIL_Handle_t railHandle) Indicate whether this chip supports SQ-based PHY.
bool	RAIL_SupportsTxPowerMode (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t powerMode, RAIL_TxPowerLevel_t *pMaxPowerLevel) Indicate whether this chip supports a particular power mode (PA).
bool	RAIL_SupportsTxPowerModeAlt (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t *powerMode, RAIL_TxPowerLevel_t *maxPowerLevel, RAIL_TxPowerLevel_t *minPowerLevel) Indicate whether this chip supports a particular power mode (PA) and provides the maximum and minimum power level for that power mode if supported by the chip.
bool	RAIL_SupportsTxToTx (RAIL_Handle_t railHandle) Indicate whether this chip supports automatic TX to TX transitions.
bool	RAIL_SupportsProtocolBLE (RAIL_Handle_t railHandle) Indicate whether RAIL supports the BLE protocol on this chip.
bool	RAIL_BLE_Supports1MbpsNonViterbi (RAIL_Handle_t railHandle) Indicate whether this chip supports BLE 1 Mbps Non-Viterbi PHY.
bool	RAIL_BLE_Supports1MbpsViterbi (RAIL_Handle_t railHandle) Indicate whether this chip supports BLE 1 Mbps Viterbi PHY.
bool	RAIL_BLE_Supports1Mbps (RAIL_Handle_t railHandle) Indicate whether this chip supports BLE 1 Mbps operation.
bool	RAIL_BLE_Supports2MbpsNonViterbi (RAIL_Handle_t railHandle) Indicate whether this chip supports BLE 2 Mbps Non-Viterbi PHY.

bool	RAIL_BLE_Supports2MbpsViterbi (RAIL_Handle_t railHandle) Indicate whether this chip supports BLE 2 Mbps Viterbi PHY.
bool	RAIL_BLE_Supports2Mbps (RAIL_Handle_t railHandle) Indicate whether this chip supports BLE 2 Mbps operation.
bool	RAIL_BLE_SupportsAntennaSwitching (RAIL_Handle_t railHandle) Indicate whether this chip supports BLE Antenna Switching needed for Angle-of-Arrival receives or Angle-of-Departure transmits.
bool	RAIL_BLE_SupportsCodedPhy (RAIL_Handle_t railHandle) Indicate whether this chip supports BLE Coded PHY used for Long-Range.
bool	RAIL_BLE_SupportsCte (RAIL_Handle_t railHandle) Indicate whether this chip supports BLE CTE (Constant Tone Extension) needed for Angle-of-Arrival/Departure transmits.
bool	RAIL_BLE_SupportsIQSampling (RAIL_Handle_t railHandle) Indicate whether this chip supports BLE IQ Sampling needed for Angle-of-Arrival/Departure receives.
bool	RAIL_BLE_SupportsPhySwitchToRx (RAIL_Handle_t railHandle) Indicate whether this chip supports BLE PHY switch to RX functionality, which is used to switch BLE PHYs at a specific time to receive auxiliary packets.
bool	RAIL_BLE_SupportsQuuppa (RAIL_Handle_t railHandle) Indicate whether this chip supports the Quuppa PHY.
bool	RAIL_BLE_SupportsSignalIdentifier (RAIL_Handle_t railHandle) Indicate whether this chip supports BLE signal identifier.
bool	RAIL_BLE_SupportsSimulscanPhy (RAIL_Handle_t railHandle) Indicate whether this chip supports BLE Simulscan PHY used for simultaneous BLE 1 Mbps and Coded PHY reception.
bool	RAIL_SupportsProtocolIEEE802154 (RAIL_Handle_t railHandle) Indicate whether this chip supports the IEEE 802.15.4 protocol.
bool	RAIL_IEEE802154_SupportsCoexPhy (RAIL_Handle_t railHandle) Indicate whether this chip supports the IEEE 802.15.4 Wi-Fi Coexistence PHY.
bool	RAIL_SupportsIEEE802154Band2P4 (RAIL_Handle_t railHandle) Indicate whether this chip supports the IEEE 802.15.4 2.4 GHz band variant.
bool	RAIL_SupportsThermalProtection (RAIL_Handle_t railHandle) Indicate whether this chip supports the thermal protection.
bool	RAIL_IEEE802154_SupportsRxChannelSwitching (RAIL_Handle_t railHandle) Indicate whether this chip supports the IEEE 802.15.4 2.4 RX channel switching.
bool	RAIL_IEEE802154_SupportsCustom1Phy (RAIL_Handle_t railHandle) Indicate whether this chip supports the IEEE 802.15.4 PHY with custom settings.
bool	RAIL_IEEE802154_SupportsFemPhy (RAIL_Handle_t railHandle) Indicate whether this chip supports the IEEE 802.15.4 front end module optimized PHY.
bool	RAIL_IEEE802154_SupportsCancelFramePendingLookup (RAIL_Handle_t railHandle) Indicate whether this chip supports canceling the frame-pending lookup event RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND when the radio transitions to a state that renders the the reporting of this event moot (i.e., too late for the stack to influence the outgoing Ack).
bool	RAIL_IEEE802154_SupportsEarlyFramePendingLookup (RAIL_Handle_t railHandle) Indicate whether this chip supports early triggering of the frame-pending lookup event RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND just after MAC address fields have been received.

bool	RAIL_IEEE802154_SupportsDualPaConfig (RAIL_Handle_t railHandle) Indicate whether RAIL supports dual PA mode on this chip.
bool	RAIL_IEEE802154_SupportsEEEnhancedAck (RAIL_Handle_t railHandle) Indicate whether this chip supports IEEE 802.15.4E-2012 Enhanced Acking.
bool	RAIL_IEEE802154_SupportsEMultipurposeFrames (RAIL_Handle_t railHandle) Indicate whether this chip supports IEEE 802.15.4E-2012 Multipurpose frame reception.
bool	RAIL_IEEE802154_SupportsESubsetGB868 (RAIL_Handle_t railHandle) Indicate whether this chip supports the IEEE 802.15.4E-2012 feature subset needed for Zigbee R22 GB868.
bool	RAIL_IEEE802154_SupportsG4ByteCrc (RAIL_Handle_t railHandle) Indicate whether this chip supports IEEE 802.15.4G-2012 reception and transmission of frames with 4-byte CRC.
bool	RAIL_IEEE802154_SupportsGDynFec (RAIL_Handle_t railHandle) Indicate whether this chip supports IEEE 802.15.4G dynamic FEC.
bool	RAIL_SupportsProtocolWiSUN (RAIL_Handle_t railHandle) Indicate whether this chip supports Wi-SUN.
bool	RAIL_IEEE802154_SupportsGModeSwitch (RAIL_Handle_t railHandle) Indicate whether this chip supports Wi-SUN mode switching.
bool	RAIL_IEEE802154_SupportsGSubsetGB868 (RAIL_Handle_t railHandle) Indicate whether this chip supports IEEE 802.15.4G-2012 feature subset needed for Zigbee R22 GB868.
bool	RAIL_IEEE802154_SupportsGUnwhitenedRx (RAIL_Handle_t railHandle) Indicate whether this chip supports IEEE 802.15.4G-2012 reception of unwhitened frames.
bool	RAIL_IEEE802154_SupportsGUnwhitenedTx (RAIL_Handle_t railHandle) Indicate whether this chip supports IEEE 802.15.4G-2012 transmission of unwhitened frames.
bool	RAIL_WMBUS_SupportsSimultaneousTCRx (RAIL_Handle_t railHandle) Indicate whether this chip supports WMBUS simultaneous M2O RX of T and C modes.
bool	RAIL_SupportsProtocolZWave (RAIL_Handle_t railHandle) Indicate whether this chip supports the Z-Wave protocol.
bool	RAIL_ZWAVE_SupportsConcPhy (RAIL_Handle_t railHandle) Indicate whether this chip supports the Z-Wave concurrent PHY.
bool	RAIL_ZWAVE_SupportsEnergyDetectPhy (RAIL_Handle_t railHandle) Indicate whether this chip supports the Z-Wave energy detect PHY.
bool	RAIL_ZWAVE_SupportsRegionPti (RAIL_Handle_t railHandle) Indicate whether this chip supports Z-Wave Region in PTI.
bool	RAIL_IEEE802154_SupportsSignalIdentifier (RAIL_Handle_t railHandle) Indicate whether this chip supports IEEE 802.15.4 signal identifier.
bool	RAIL_SupportsFastRx2Rx (RAIL_Handle_t railHandle) Indicate whether this chip supports fast RX-to-RX.
bool	RAIL_SupportsCollisionDetection (RAIL_Handle_t railHandle) Indicate whether this chip supports collision detection.
bool	RAIL_SupportsProtocolSidewalk (RAIL_Handle_t railHandle) Indicate whether this chip supports Sidewalk protocol.
bool	RAIL_SupportsTrustZoneSecurePeripherals (RAIL_Handle_t railHandle) Indicate whether this chip supports TrustZone secure configuration of peripherals used by RAIL.

- bool [RAIL_SupportsPrsLnaBypass](#)(RAIL_Handle_t railHandle)
Indicate whether this chip supports automatic PRS LNA bypass for external FEM.
- bool [RAIL_SupportsProtocolBTC](#)(RAIL_Handle_t railHandle)
Indicate whether RAIL supports the BTC protocol on this chip.

Macros

- #define** RAIL_SUPPORTS_DUAL_BAND 1
Boolean to indicate whether the selected chip supports both Sub-GHz and 2.4 GHz bands.
- #define** RAIL_FEAT_DUAL_BAND_RADIO RAIL_SUPPORTS_DUAL_BAND
Backwards-compatible synonym of [RAIL_SUPPORTS_DUAL_BAND](#).
- #define** RAIL_SUPPORTS_2P4GHZ_BAND 1
Boolean to indicate whether the selected chip supports the 2.4 GHz band.
- #define** RAIL_FEAT_2G4_RADIO RAIL_SUPPORTS_2P4GHZ_BAND
Backwards-compatible synonym of [RAIL_SUPPORTS_2P4GHZ_BAND](#).
- #define** RAIL_SUPPORTS_SUBGHZ_BAND 1
Boolean to indicate whether the selected chip supports Sub-GHz bands.
- #define** RAIL_FEAT_SUBGIG_RADIO RAIL_SUPPORTS_SUBGHZ_BAND
Backwards-compatible synonym of [RAIL_SUPPORTS_SUBGHZ_BAND](#).
- #define** RAIL_SUPPORTS_OFDM_PA 0
Boolean to indicate whether the selected chip supports OFDM PA.
- #define** RAIL_SUPPORTS_ADDR_FILTER_ADDRESS_BIT_MASK 0
Boolean to indicate whether the selected chip supports bit masked address filtering.
- #define** RAIL_SUPPORTS_ADDR_FILTER_MASK 1
Boolean to indicate whether the selected chip supports address filter mask information for incoming packets in [RAIL_RxPacketInfo_t::filterMask](#) and [RAIL_IEEE802154_Address_t::filterMask](#).
- #define** RAIL_SUPPORTS_ALTERNATE_TX_POWER 0
Boolean to indicate whether the selected chip supports alternate power settings for the Power Amplifier.
- #define** RAIL_FEAT_ALTERNATE_POWER_TX_SUPPORTED RAIL_SUPPORTS_ALTERNATE_TX_POWER
Backwards-compatible synonym of [RAIL_SUPPORTS_ALTERNATE_TX_POWER](#).
- #define** RAIL_SUPPORTS_ANTENNA_DIVERSITY 0
Boolean to indicate whether the selected chip supports antenna diversity.
- #define** RAIL_FEAT_ANTENNA_DIVERSITY RAIL_SUPPORTS_ANTENNA_DIVERSITY
Backwards-compatible synonym of [RAIL_SUPPORTS_ANTENNA_DIVERSITY](#).
- #define** RAIL_SUPPORTS_PATH_DIVERSITY 0
Boolean to indicate whether the selected chip supports internal RF path diversity.
- #define** RAIL_SUPPORTS_CHANNEL_HOPPING 0
Boolean to indicate whether the selected chip supports channel hopping.
- #define** RAIL_FEAT_CHANNEL_HOPPING RAIL_SUPPORTS_CHANNEL_HOPPING
Backwards-compatible synonym of [RAIL_SUPPORTS_CHANNEL_HOPPING](#).
- #define** RAIL_SUPPORTS_DUAL_SYNC_WORDS 1
Boolean to indicate whether the selected chip supports dual sync words.

#define	RAIL_SUPPORTS_TX_TO_TX 1	Boolean to indicate whether the selected chip supports automatic transitions from TX to TX.
#define	RAIL_SUPPORTS_TX_REPEAT_START_TO_START 0	Boolean to indicate whether the selected chip supports RAIL_TX_REPEAT_OPTION_START_TO_START .
#define	RAIL_SUPPORTS_EXTERNAL_THERMISTOR 0	Boolean to indicate whether the selected chip supports thermistor measurements.
#define	RAIL_FEAT_EXTERNAL_THERMISTOR RAIL_SUPPORTS_EXTERNAL_THERMISTOR	Backwards-compatible synonym of RAIL_SUPPORTS_EXTERNAL_THERMISTOR .
#define	RAIL_SUPPORTS_HFXO_COMPENSATION 0	Boolean to indicate whether the selected chip supports HFXO compensation.
#define	RAIL_SUPPORTS_AUXADC 0	Boolean to indicate whether the selected chip supports AUXADC measurements.
#define	RAIL_SUPPORTS_PRECISION_LFRCO 0	Boolean to indicate whether the selected chip supports a high-precision LFRCO.
#define	RAIL_SUPPORTS_RADIO_ENTROPY 1	Boolean to indicate whether the selected chip supports radio entropy.
#define	RAIL_SUPPORTS_RFSENSE_ENERGY_DETECTION 0	Boolean to indicate whether the selected chip supports RF Sense Energy Detection Mode.
#define	RAIL_SUPPORTS_RFSENSE_SELECTIVE_OOK 0	Boolean to indicate whether the selected chip supports RF Sense Selective(OOK) Mode.
#define	RAIL_FEAT_RFSENSE_SELECTIVE_OOK_MODE_SUPPORTED RAIL_SUPPORTS_RFSENSE_SELECTIVE_OOK	Backwards-compatible synonym of RAIL_SUPPORTS_RFSENSE_SELECTIVE_OOK .
#define	RAIL_SUPPORTS_VDET 0	Boolean to indicate whether the selected chip supports the VDET voltage measurement feature.
#define	RAIL_SUPPORTS_PROTOCOL_BLE RAIL_SUPPORTS_2P4GHZ_BAND	Boolean to indicate whether the selected chip supports BLE.
#define	RAIL_BLE_SUPPORTS_1MBPS_NON_VITERBI 0	Boolean to indicate whether the selected chip supports BLE 1 Mbps Non-Viterbi PHY.
#define	RAIL_BLE_SUPPORTS_1MBPS_VITERBI RAIL_SUPPORTS_PROTOCOL_BLE	Boolean to indicate whether the selected chip supports BLE 1 Mbps Viterbi PHY.
#define	RAIL_BLE_SUPPORTS_1MBPS (RAIL_BLE_SUPPORTS_1MBPS_NON_VITERBI RAIL_BLE_SUPPORTS_1MBPS_VITERBI)	Boolean to indicate whether the selected chip supports BLE 1 Mbps operation.
#define	RAIL_BLE_SUPPORTS_2MBPS_NON_VITERBI 0	Boolean to indicate whether the selected chip supports BLE 2 Mbps Non-Viterbi PHY.
#define	RAIL_BLE_SUPPORTS_2MBPS_VITERBI RAIL_SUPPORTS_PROTOCOL_BLE	Boolean to indicate whether the selected chip supports BLE 2 Mbps Viterbi PHY.
#define	RAIL_BLE_SUPPORTS_2MBPS (RAIL_BLE_SUPPORTS_2MBPS_NON_VITERBI RAIL_BLE_SUPPORTS_2MBPS_VITERBI)	Boolean to indicate whether the selected chip supports BLE 2 Mbps operation.
#define	RAIL_BLE_SUPPORTS_ANTENNA_SWITCHING 0	Boolean to indicate whether the selected chip supports BLE Antenna Switching needed for Angle-of-Arrival receives or Angle-of-Departure transmits.

#define	RAIL_BLE_SUPPORTS_CODED_PHY 0	Boolean to indicate whether the selected chip supports the BLE Coded PHY used for Long-Range.
#define	RAIL_FEAT_BLE_CODED RAIL_BLE_SUPPORTS_CODED_PHY	Backwards-compatible synonym of RAIL_BLE_SUPPORTS_CODED_PHY .
#define	RAIL_BLE_SUPPORTS_SIMULSCAN_PHY 0	Boolean to indicate whether the selected chip supports the BLE Simulscan PHY used for simultaneous BLE 1 Mbps and Coded PHY reception.
#define	RAIL_BLE_SUPPORTS_CTE 0	Boolean to indicate whether the selected chip supports BLE CTE (Constant Tone Extension) needed for Angle-of-Arrival/Departure transmits.
#define	RAIL_BLE_SUPPORTS_QUUPPA 0	Boolean to indicate whether the selected chip supports the Quuppa PHY.
#define	RAIL_BLE_SUPPORTS_IQ_SAMPLING 0	Boolean to indicate whether the selected chip supports BLE IQ Sampling needed for Angle-of-Arrival/Departure receives.
#define	RAIL_BLE_SUPPORTS_AOX undefined	Boolean to indicate whether the selected chip supports some BLE AOX features.
#define	RAIL_FEAT_BLE_AOX_SUPPORTED RAIL_BLE_SUPPORTS_AOX	Backwards-compatible synonym of RAIL_BLE_SUPPORTS_AOX .
#define	RAIL_BLE_SUPPORTS_PHY_SWITCH_TO_RX RAIL_SUPPORTS_PROTOCOL_BLE	Boolean to indicate whether the selected chip supports BLE PHY switch to RX functionality, which is used to switch BLE PHYs at a specific time to receive auxiliary packets.
#define	RAIL_FEAT_BLE_PHY_SWITCH_TO_RX RAIL_BLE_SUPPORTS_PHY_SWITCH_TO_RX	Backwards-compatible synonym of RAIL_BLE_SUPPORTS_PHY_SWITCH_TO_RX .
#define	RAIL_SUPPORTS_PROTOCOL_IEEE802154 1	Boolean to indicate whether the selected chip supports IEEE 802.15.4.
#define	RAIL_IEEE802154_SUPPORTS_COEX_PHY 0	Boolean to indicate whether the selected chip supports the 802.15.4 Wi-Fi Coexistence PHY.
#define	RAIL_FEAT_802154_COEX_PHY RAIL_IEEE802154_SUPPORTS_COEX_PHY	Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_COEX_PHY .
#define	RAIL_SUPPORTS_IEEE802154_BAND_2P4 (RAIL_SUPPORTS_PROTOCOL_IEEE802154 && RAIL_SUPPORTS_2P4GHZ_BAND)	Boolean to indicate whether the selected chip supports the IEEE 802.15.4 2.4 GHz band variant.
#define	RAIL_IEEE802154_SUPPORTS_RX_CHANNEL_SWITCHING 0	Boolean to indicate whether the selected chip supports the IEEE 802.15.4 2.4 RX channel switching.
#define	RAIL_IEEE802154_SUPPORTS_FEM_PHY (RAIL_SUPPORTS_IEEE802154_BAND_2P4)	Boolean to indicate whether the selected chip supports a front end module.
#define	RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868 RAIL_SUPPORTS_PROTOCOL_IEEE802154	Boolean to indicate whether the selected chip supports IEEE 802.15.4E-2012 feature subset needed for Zigbee R22 GB868.
#define	RAIL_FEAT_IEEE802154_E_GB868_SUPPORTED RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868	Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868 .
#define	RAIL_IEEE802154_SUPPORTS_E_ENHANCED_ACK RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868	Boolean to indicate whether the selected chip supports IEEE 802.15.4E-2012 Enhanced ACKing.

#define	RAIL_FEAT_IEEE802154_E_ENH_ACK_SUPPORTED RAIL_IEEE802154_SUPPORTS_E_ENHANCED_ACK Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_E_ENHANCED_ACK .
#define	RAIL_IEEE802154_SUPPORTS_E_MULTIPURPOSE_FRAMES RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868 Boolean to indicate whether the selected chip supports receiving IEEE 802.15.4E-2012 Multipurpose frames.
#define	RAIL_FEAT_IEEE802154_MULTIPURPOSE_FRAME_SUPPORTED RAIL_IEEE802154_SUPPORTS_E_MULTIPURPOSE_FRAMES Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_E_MULTIPURPOSE_FRAMES .
#define	RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868 ((RAIL_SUPPORTS_PROTOCOL_IEEE802154 != 0) && (RAIL_SUPPORTS_SUBGHZ_BAND != 0)) Boolean to indicate whether the selected chip supports IEEE 802.15.4G-2012 feature subset needed for Zigbee R22 GB868.
#define	RAIL_FEAT_IEEE802154_G_GB868_SUPPORTED RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868 Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868 .
#define	RAIL_IEEE802154_SUPPORTS_G_DYNFEC 0 Boolean to indicate whether the selected chip supports dynamic FEC See also runtime refinement RAIL_IEEE802154_SupportsGDynFec() .
#define	RAIL_IEEE802154_SUPPORTS_G_MODESWITCH 0 Boolean to indicate whether the selected chip supports Wi-SUN mode switching See also runtime refinement RAIL_IEEE802154_SupportsGModeSwitch() .
#define	RAIL_IEEE802154_SUPPORTS_G_4BYTE_CRC RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868 Boolean to indicate whether the selected chip supports IEEE 802.15.4G-2012 reception and transmission of frames with 4-byte CRC.
#define	RAIL_FEAT_IEEE802154_G_4BYTE_CRC_SUPPORTED RAIL_IEEE802154_SUPPORTS_G_4BYTE_CRC Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_G_4BYTE_CRC .
#define	RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_RX RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868 Boolean to indicate whether the selected chip supports IEEE 802.15.4G-2012 reception of unwhitened frames.
#define	RAIL_FEAT_IEEE802154_G_UNWHITENED_RX_SUPPORTED RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_RX Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_RX .
#define	RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_TX RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868 Boolean to indicate whether the selected chip supports IEEE 802.15.4G-2012 transmission of unwhitened frames.
#define	RAIL_FEAT_IEEE802154_G_UNWHITENED_TX_SUPPORTED RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_TX Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_TX .
#define	RAIL_IEEE802154_SUPPORTS_CANCEL_FRAME_PENDING_LOOKUP RAIL_SUPPORTS_PROTOCOL_IEEE802154 Boolean to indicate whether the selected chip supports canceling the frame-pending lookup event RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND when the radio transitions to a state that renders the the reporting of this event moot (i.e., too late for the stack to influence the outgoing ACK).
#define	RAIL_FEAT_IEEE802154_CANCEL_FP_LOOKUP_SUPPORTED RAIL_IEEE802154_SUPPORTS_CANCEL_FRAME_PENDING_LOOKUP Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_CANCEL_FRAME_PENDING_LOOKUP .
#define	RAIL_IEEE802154_SUPPORTS_EARLY_FRAME_PENDING_LOOKUP RAIL_SUPPORTS_PROTOCOL_IEEE802154 Boolean to indicate whether the selected chip supports early triggering of the frame-pending lookup event RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND just after MAC address fields have been received.
#define	RAIL_FEAT_IEEE802154_EARLY_FP_LOOKUP_SUPPORTED RAIL_IEEE802154_SUPPORTS_EARLY_FRAME_PENDING_LOOKUP Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_EARLY_FRAME_PENDING_LOOKUP .

#define	RAIL_IEEE802154_SUPPORTS_DUAL_PA_CONFIG 0	Boolean to indicate whether the selected chip supports dual PA configs for mode switch or concurrent mode.
#define	RAIL_SUPPORTS_DBM_POWERSETTING_MAPPING_TABLE 0	Boolean to indicate whether the selected chip supports the pa power setting table.
#define	RAIL_IEEE802154_SUPPORTS_CUSTOM1_PHY 0	Boolean to indicate whether the selected chip supports IEEE 802.15.4 PHY with custom settings.
#define	RAIL_SUPPORTS_PROTOCOL_WI_SUN 0	Boolean to indicate whether the selected chip supports Wi-SUN See also runtime refinement RAIL_SupportsProtocolWISUN() .
#define	RAIL_WMBUS_SUPPORTS_SIMULTANEOUS_T_C_RX 0	Boolean to indicate whether the selected chip supports WMBUS simultaneous M2O RX of T and C modes set by RAIL_WMBUS_Config() .
#define	RAIL_SUPPORTS_PROTOCOL_ZWAVE 0	Boolean to indicate whether the selected chip supports Z-Wave.
#define	RAIL_FEAT_ZWAVE_SUPPORTED RAIL_SUPPORTS_PROTOCOL_ZWAVE	Backwards-compatible synonym of RAIL_SUPPORTS_PROTOCOL_ZWAVE .
#define	RAIL_ZWAVE_SUPPORTS_ED_PHY 0	Boolean to indicate whether the selected chip supports energy detect PHY.
#define	RAIL_ZWAVE_SUPPORTS_CONC_PHY 0	Boolean to indicate whether the selected chip supports concurrent PHY.
#define	RAIL_SUPPORTS_SQ_PHY 0	Boolean to indicate whether the selected chip supports SQ-based PHY.
#define	RAIL_ZWAVE_SUPPORTS_REGION_PTI RAIL_SUPPORTS_PROTOCOL_ZWAVE	Boolean to indicate whether the code supports Z-Wave region information in PTI and newer RAIL_ZWAVE_RegionConfig_t structure See also runtime refinement RAIL_ZWAVE_SupportsRegionPti() .
#define	RAIL_FEAT_ZWAVE_REGION_PTI RAIL_ZWAVE_SUPPORTS_REGION_PTI	Backwards-compatible synonym of RAIL_ZWAVE_SUPPORTS_REGION_PTI .
#define	RAIL_SUPPORTS_RX_RAW_DATA 1	Boolean to indicate whether the selected chip supports raw RX data sources other than RAIL_RxDataSource_t::RX_PACKET_DATA .
#define	RAIL_SUPPORTS_DIRECT_MODE 0	Boolean to indicate whether the selected chip supports direct mode.
#define	RAIL_SUPPORTS_RX_DIRECT_MODE_DATA_TO_FIFO 0	Boolean to indicate whether the selected chip supports RX direct mode data to FIFO.
#define	RAIL_SUPPORTS_MFM 0	Boolean to indicate whether the selected chip supports MFM protocol.
#define	RAIL_IEEE802154_SUPPORTS_SIGNAL_IDENTIFIER 0	Boolean to indicate whether the selected chip supports 802.15.4 signal detection.
#define	RAIL_BLE_SUPPORTS_SIGNAL_IDENTIFIER 0	Boolean to indicate whether the selected chip supports BLE signal detection.
#define	RAIL_SUPPORTS_PROTOCOL_BTC 0	Boolean to indicate whether the selected chip supports BTC.

#define	RAIL_SUPPORTS_RSSI_DETECT_THRESHOLD (0U) Boolean to indicate whether the selected chip supports configurable RSSI threshold set by RAIL_SetRssiDetectThreshold() .
#define	RAIL_SUPPORTS_THERMAL_PROTECTION (0U) Boolean to indicate whether the selected chip supports thermal protection set by RAIL_ConfigThermalProtection() .
#define	RAIL_SUPPORTS_FAST_RX2RX (0U) Boolean to indicate whether the selected chip supports fast RX-to-RX enabled by RAIL_RX_OPTION_FAST_RX2RX .
#define	RAIL_SUPPORTS_COLLISION_DETECTION (0U) Boolean to indicate whether the selected chip supports collision detection enabled by RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION See also runtime refinement RAIL_SupportsCollisionDetection() .
#define	RAIL_SUPPORTS_PROTOCOL_SIDEWALK (0U) Boolean to indicate whether the selected chip supports Sidewalk protocol.
#define	RAIL_SUPPORTS_TRUSTZONE_SECURE_PERIPHERALS (0U) Boolean to indicate whether the selected chip supports TrustZone secure configuration of peripherals used by RAIL.
#define	RAIL_SUPPORTS_PRS_LNA_BYPASS (0U) Boolean to indicate whether the selected chip supports automatic PRS LNA bypass for external FEM.

Function Documentation

RAIL_Supports2p4GHzBand

```
bool RAIL_Supports2p4GHzBand (RAIL\_Handle\_t railHandle)
```

Indicate whether RAIL supports 2.4 GHz band operation on this chip.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if the 2.4 GHz band is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_2P4GHZ_BAND](#).

Definition at line 7109 of file `common/rail.h`

RAIL_SupportsSubGHzBand

```
bool RAIL_SupportsSubGHzBand (RAIL\_Handle\_t railHandle)
```

Indicate whether RAIL supports SubGHz band operation on this chip.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if the SubGHz band is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_SUBGHZ_BAND](#).

Definition at line 7119 of file `common/rail.h`

RAIL_SupportsDualBand

```
bool RAIL_SupportsDualBand (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports dual 2.4 GHz and SubGHz band operation.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if the dual band is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_DUAL_BAND](#).

Definition at line 7129 of file common/rail.h

RAIL_SupportsAddrFilterAddressBitMask

```
bool RAIL_SupportsAddrFilterAddressBitMask (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports bit masked address filtering.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if bit masked address filtering is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_ADDR_FILTER_ADDRESS_BIT_MASK](#).

Definition at line 7140 of file common/rail.h

RAIL_SupportsAddrFilterMask

```
bool RAIL_SupportsAddrFilterMask (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports address filter mask information for incoming packets in [RAIL_RxPacketInfo_t::filterMask](#) and [RAIL_IEEE802154_Address_t::filterMask](#).

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if address filter information is supported; false otherwise (in which case [RAIL_RxPacketInfo_t::filterMask](#) value is undefined).
- Runtime refinement of compile-time [RAIL_SUPPORTS_ADDR_FILTER_MASK](#).

Definition at line 7154 of file common/rail.h

RAIL_SupportsAlternateTxPower

```
bool RAIL_SupportsAlternateTxPower (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports alternate TX power settings.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if alternate TX power settings are supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_ALTERNATE_TX_POWER](#).

Definition at line 7165 of file common/rail.h

RAIL_SupportsAntennaDiversity

```
bool RAIL_SupportsAntennaDiversity (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports antenna diversity.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if antenna diversity is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_ANTENNA_DIVERSITY](#).

Note

- Certain radio configurations may not support this feature even if the chip in general claims to support it.

Definition at line 7178 of file common/rail.h

RAIL_SupportsPathDiversity

```
bool RAIL_SupportsPathDiversity (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports internal RF path diversity.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if RF path diversity is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_PATH_DIVERSITY](#).

Note

- Certain radio configurations may not support this feature even if the chip in general claims to support it.

Definition at line 7191 of file common/rail.h

RAIL_SupportsAuxAdc

```
bool RAIL_SupportsAuxAdc (RAIL_Handle_t railHandle)
```

Indicate whether RAIL supports AUXADC measurements on this chip.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if AUXADC measurements are supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_AUXADC](#).

Definition at line 7201 of file common/rail.h

RAIL_SupportsChannelHopping

```
bool RAIL_SupportsChannelHopping (RAIL_Handle_t railHandle)
```

Indicate whether RAIL supports channel hopping on this chip.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if channel hopping is supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_CHANNEL_HOPPING](#).

Definition at line 7211 of file common/rail.h

RAIL_SupportsDirectMode

```
bool RAIL_SupportsDirectMode (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports direct mode.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if direct mode is supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_DIRECT_MODE](#).

Definition at line 7222 of file common/rail.h

RAIL_SupportsDualSyncWords

```
bool RAIL_SupportsDualSyncWords (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports dual sync words.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if dual sync words are supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_DUAL_SYNC_WORDS](#).

Note

- Certain radio configurations may not support this feature even if the chip in general claims to support it.

Definition at line 7235 of file common/rail.h

RAIL_SupportsTxRepeatStartToStart

```
bool RAIL_SupportsTxRepeatStartToStart (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports start to start TX repeats.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if start to start TX repeats are supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_TX_REPEAT_START_TO_START](#).

Definition at line 7246 of file common/rail.h

RAIL_SupportsVdet

```
bool RAIL_SupportsVdet (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports VDET.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if VDET is supported; false otherwise.

Definition at line 7254 of file common/rail.h

RAIL_SupportsExternalThermistor

bool RAIL_SupportsExternalThermistor ([RAIL_Handle_t](#) railHandle)

Indicate whether RAIL supports thermistor measurements on this chip.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if thermistor measurements are supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_EXTERNAL_THERMISTOR](#).

Definition at line 7264 of file `common/rail.h`

RAIL_SupportsHFXOCompensation

bool RAIL_SupportsHFXOCompensation ([RAIL_Handle_t](#) railHandle)

Indicate whether RAIL supports HFXO compensation on this chip.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if HFXO compensation is supported and [RAIL_ConfigHFXOThermistor\(\)](#) has been successfully called; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_HFXO_COMPENSATION](#).

Definition at line 7276 of file `common/rail.h`

RAIL_SupportsMfm

bool RAIL_SupportsMfm ([RAIL_Handle_t](#) railHandle)

Indicate whether this chip supports MFM protocol.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if MFM protocol is supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_MFM](#).

Definition at line 7286 of file `common/rail.h`

RAIL_SupportsOFDMPA

bool RAIL_SupportsOFDMPA ([RAIL_Handle_t](#) railHandle)

Indicate whether RAIL supports OFDM band operation on this chip.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if OFDM operation is supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_OFDM_PA](#).

Definition at line 7296 of file common/rail.h

RAIL_SupportsPrecisionLFRCO

```
bool RAIL_SupportsPrecisionLFRCO (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports a high-precision LFRCO.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if high-precision LFRCO is supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_PRECISION_LFRCO](#).

Definition at line 7306 of file common/rail.h

RAIL_SupportsRadioEntropy

```
bool RAIL_SupportsRadioEntropy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports radio entropy.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if radio entropy is supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_RADIO_ENTROPY](#).

Definition at line 7316 of file common/rail.h

RAIL_SupportsRfSenseEnergyDetection

```
bool RAIL_SupportsRfSenseEnergyDetection (RAIL_Handle_t railHandle)
```

Indicate whether RAIL supports RF Sense Energy Detection Mode on this chip.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if RF Sense Energy Detection Mode is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_RFSENSE_ENERGY_DETECTION](#).

Definition at line 7327 of file `common/rail.h`

RAIL_SupportsRfSenseSelectiveOok

```
bool RAIL_SupportsRfSenseSelectiveOok (RAIL_Handle_t railHandle)
```

Indicate whether RAIL supports RF Sense Selective(OOK) Mode on this chip.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if RF Sense Selective(OOK) Mode is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_RFSENSE_SELECTIVE_OOK](#).

Definition at line 7337 of file `common/rail.h`

RAIL_SupportsRssiDetectThreshold

```
bool RAIL_SupportsRssiDetectThreshold (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports configurable RSSI threshold set by [RAIL_SetRssiDetectThreshold\(\)](#).

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if setting configurable RSSI is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_RSSI_DETECT_THRESHOLD](#).

Definition at line 7348 of file `common/rail.h`

RAIL_SupportsRxDirectModeDataToFifo

```
bool RAIL_SupportsRxDirectModeDataToFifo (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports RX direct mode data to FIFO.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

true if direct mode data to FIFO is supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_RX_DIRECT_MODE_DATA_TO_FIFO](#).

Definition at line 7359 of file `common/rail.h`

RAIL_SupportsRxRawData

```
bool RAIL_SupportsRxRawData (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports raw RX data sources other than [RAIL_RxDataSource_t::RX_PACKET_DATA](#).

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if direct mode is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_RX_RAW_DATA](#).

Definition at line 7370 of file `common/rail.h`

RAIL_SupportsSQPhy

```
bool RAIL_SupportsSQPhy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports SQ-based PHY.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if the SQ-based PHY is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_SQ_PHY](#).

Definition at line 7380 of file `common/rail.h`

RAIL_SupportsTxPowerMode

```
bool RAIL_SupportsTxPowerMode (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t powerMode, RAIL_TxPowerLevel_t * pMaxPowerLevel)
```

Indicate whether this chip supports a particular power mode (PA).

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.
[in]	powerMode	The power mode to check if supported.
[out]	pMaxPowerLevel	A pointer to a RAIL_TxPowerLevel_t that if non-NULL will be filled in with the power mode's highest power level allowed if this function returns true.

Returns

- true if the powerMode is supported; false otherwise.

This function has no compile-time equivalent.

Note

- Consider using [RAIL_SupportsTxPowerModeAlt](#) to also get the power mode's lowest allowed power level.

Definition at line 7399 of file `common/rail.h`

RAIL_SupportsTxPowerModeAlt

```
bool RAIL_SupportsTxPowerModeAlt (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t * powerMode, RAIL_TxPowerLevel_t * maxPowerLevel, RAIL_TxPowerLevel_t * minPowerLevel)
```

Indicate whether this chip supports a particular power mode (PA) and provides the maximum and minimum power level for that power mode if supported by the chip.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.
[inout]	powerMode	A pointer to PA power mode to check if supported. If RAIL_TX_POWER_MODE_2P4GIG_HIGHEST or RAIL_TX_POWER_MODE_SUBGIG_HIGHEST is passed in, it will be updated to the highest corresponding PA available on the chip.
[out]	maxPowerLevel	A pointer to a RAIL_TxPowerLevel_t that if non-NULL will be filled in with the power mode's highest power level allowed if this function returns true.
[out]	minPowerLevel	A pointer to a RAIL_TxPowerLevel_t that if non-NULL will be filled in with the power mode's lowest power level allowed if this function returns true.

Returns

- true if powerMode is supported; false otherwise.

This function has no compile-time equivalent.

Definition at line 7423 of file `common/rail.h`

RAIL_SupportsTxToTx

```
bool RAIL_SupportsTxToTx (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports automatic TX to TX transitions.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if TX to TX transitions are supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_TX_TO_TX](#).

Definition at line 7438 of file `common/rail.h`

RAIL_SupportsProtocolBLE

```
bool RAIL_SupportsProtocolBLE (RAIL_Handle_t railHandle)
```

Indicate whether RAIL supports the BLE protocol on this chip.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if BLE is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_PROTOCOL_BLE](#).

Definition at line 7460 of file common/rail.h

RAIL_BLE_Supports1MbpsNonViterbi

```
bool RAIL_BLE_Supports1MbpsNonViterbi (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE 1 Mbps Non-Viterbi PHY.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if BLE 1 Mbps Non-Viterbi is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_BLE_SUPPORTS_1MBPS_NON_VITERBI](#).

Definition at line 7470 of file common/rail.h

RAIL_BLE_Supports1MbpsViterbi

```
bool RAIL_BLE_Supports1MbpsViterbi (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE 1 Mbps Viterbi PHY.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if BLE 1 Mbps Viterbi is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_BLE_SUPPORTS_1MBPS_VITERBI](#).

Definition at line 7480 of file common/rail.h

RAIL_BLE_Supports1Mbps

```
bool RAIL_BLE_Supports1Mbps (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE 1 Mbps operation.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if BLE 1 Mbps operation is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_BLE_SUPPORTS_1MBPS](#).

Definition at line 7490 of file common/rail.h

RAIL_BLE_Supports2MbpsNonViterbi

```
bool RAIL_BLE_Supports2MbpsNonViterbi (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE 2 Mbps Non-Viterbi PHY.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if BLE 2 Mbps Non-Viterbi is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_BLE_SUPPORTS_2MBPS_NON_VITERBI](#).

Definition at line 7500 of file common/rail.h

RAIL_BLE_Supports2MbpsViterbi

```
bool RAIL_BLE_Supports2MbpsViterbi (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE 2 Mbps Viterbi PHY.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if BLE 2 Mbps Viterbi is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_BLE_SUPPORTS_2MBPS_VITERBI](#).

Definition at line 7510 of file common/rail.h

RAIL_BLE_Supports2Mbps

```
bool RAIL_BLE_Supports2Mbps (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE 2 Mbps operation.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if BLE 2 Mbps operation is supported; false otherwise.

Runtime refinement of compile-time [RAIL_BLE_SUPPORTS_2MBPS](#).

Definition at line 7520 of file common/rail.h

RAIL_BLE_SupportsAntennaSwitching

```
bool RAIL_BLE_SupportsAntennaSwitching (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE Antenna Switching needed for Angle-of-Arrival receives or Angle-of-Departure transmits.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if BLE Antenna Switching is supported; false otherwise.

Runtime refinement of compile-time [RAIL_BLE_SUPPORTS_ANTENNA_SWITCHING](#).

Definition at line 7531 of file common/rail.h

RAIL_BLE_SupportsCodedPhy

```
bool RAIL_BLE_SupportsCodedPhy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE Coded PHY used for Long-Range.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if BLE Coded PHY is supported; false otherwise.

Runtime refinement of compile-time [RAIL_BLE_SUPPORTS_CODED_PHY](#).

Definition at line 7541 of file common/rail.h

RAIL_BLE_SupportsCte

```
bool RAIL_BLE_SupportsCte (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE CTE (Constant Tone Extension) needed for Angle-of-Arrival/Departure transmits.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if BLE CTE is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_BLE_SUPPORTS_CTE](#).

Definition at line 7552 of file common/rail.h

RAIL_BLE_SupportsIQSampling

bool RAIL_BLE_SupportsIQSampling ([RAIL_Handle_t](#) railHandle)

Indicate whether this chip supports BLE IQ Sampling needed for Angle-of-Arrival/Departure receives.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if BLE IQ Sampling is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_BLE_SUPPORTS_IQ_SAMPLING](#).

Definition at line 7575 of file common/rail.h

RAIL_BLE_SupportsPhySwitchToRx

bool RAIL_BLE_SupportsPhySwitchToRx ([RAIL_Handle_t](#) railHandle)

Indicate whether this chip supports BLE PHY switch to RX functionality, which is used to switch BLE PHYs at a specific time to receive auxiliary packets.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if BLE PHY switch to RX is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_BLE_SUPPORTS_PHY_SWITCH_TO_RX](#).

Definition at line 7587 of file common/rail.h

RAIL_BLE_SupportsQuuppa

bool RAIL_BLE_SupportsQuuppa ([RAIL_Handle_t](#) railHandle)

Indicate whether this chip supports the Quuppa PHY.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if the Quuppa is supported; false otherwise.

Runtime refinement of compile-time [RAIL_BLE_SUPPORTS_QUUPPA](#).

Definition at line 7597 of file `common/rail.h`

RAIL_BLE_SupportsSignalIdentifier

```
bool RAIL_BLE_SupportsSignalIdentifier (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE signal identifier.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if signal identifier is supported; false otherwise.

Definition at line 7605 of file `common/rail.h`

RAIL_BLE_SupportsSimulscanPhy

```
bool RAIL_BLE_SupportsSimulscanPhy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE Simulscan PHY used for simultaneous BLE 1 Mbps and Coded PHY reception.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if BLE Simulscan PHY is supported; false otherwise.

Runtime refinement of compile-time [RAIL_BLE_SUPPORTS_SIMULSCAN_PHY](#).

Definition at line 7616 of file `common/rail.h`

RAIL_SupportsProtocolIEEE802154

```
bool RAIL_SupportsProtocolIEEE802154 (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the IEEE 802.15.4 protocol.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if the 802.15.4 protocol is supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_PROTOCOL_IEEE802154](#).

Definition at line 7626 of file common/rail.h

RAIL_IEEE802154_SupportsCoexPhy

```
bool RAIL_IEEE802154_SupportsCoexPhy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the IEEE 802.15.4 Wi-Fi Coexistence PHY.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if the 802.15.4 COEX PHY is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_COEX_PHY](#).

Definition at line 7648 of file common/rail.h

RAIL_SupportsIEEE802154Band2P4

```
bool RAIL_SupportsIEEE802154Band2P4 (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the IEEE 802.15.4 2.4 GHz band variant.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if IEEE 802.15.4 2.4 GHz band variant is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_IEEE802154_BAND_2P4](#).

Definition at line 7659 of file common/rail.h

RAIL_SupportsThermalProtection

```
bool RAIL_SupportsThermalProtection (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the thermal protection.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if thermal protection is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_THERMAL_PROTECTION](#).

Definition at line 7670 of file common/rail.h

RAIL_IEEE802154_SupportsRxChannelSwitching

```
bool RAIL_IEEE802154_SupportsRxChannelSwitching (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the IEEE 802.15.4 2.4 RX channel switching.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if IEEE 802.15.4 2.4 GHz RX channel switching is supported; false otherwise.

Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_RX_CHANNEL_SWITCHING](#).

Definition at line 7681 of file common/rail.h

RAIL_IEEE802154_SupportsCustom1Phy

```
bool RAIL_IEEE802154_SupportsCustom1Phy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the IEEE 802.15.4 PHY with custom settings.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if the 802.15.4 PHY with custom settings is supported; false otherwise.

Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_CUSTOM1_PHY](#).

Definition at line 7691 of file common/rail.h

RAIL_IEEE802154_SupportsFemPhy

```
bool RAIL_IEEE802154_SupportsFemPhy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the IEEE 802.15.4 front end module optimized PHY.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if a front end module is supported; false otherwise.

Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_FEM_PHY](#).

Definition at line 7702 of file common/rail.h

RAIL_IEEE802154_SupportsCancelFramePendingLookup

```
bool RAIL_IEEE802154_SupportsCancelFramePendingLookup (RAIL_Handle_t railHandle)
```


Indicate whether this chip supports canceling the frame-pending lookup event [RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#) when the radio transitions to a state that renders the the reporting of this event moot (i.e., too late for the stack to influence the outgoing Ack).

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if canceling the lookup event is supported; false otherwise.

Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_CANCEL_FRAME_PENDING_LOOKUP](#).

Definition at line 7716 of file common/rail.h

RAIL_IEEE802154_SupportsEarlyFramePendingLookup

```
bool RAIL_IEEE802154_SupportsEarlyFramePendingLookup (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports early triggering of the frame-pending lookup event [RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#) just after MAC address fields have been received.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if early triggering is supported; false otherwise.

Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_EARLY_FRAME_PENDING_LOOKUP](#).

Definition at line 7729 of file common/rail.h

RAIL_IEEE802154_SupportsDualPaConfig

```
bool RAIL_IEEE802154_SupportsDualPaConfig (RAIL_Handle_t railHandle)
```

Indicate whether RAIL supports dual PA mode on this chip.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if the dual PA mode is supported; false otherwise.

Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_DUAL_PA_CONFIG](#).

Definition at line 7739 of file common/rail.h

RAIL_IEEE802154_SupportsEEenhancedAck

```
bool RAIL_IEEE802154_SupportsEEenhancedAck (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports IEEE 802.15.4E-2012 Enhanced Acking.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if 802.15.4E Enhanced Acking is supported; false otherwise.

Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_E_ENHANCED_ACK](#).

Definition at line 7750 of file common/rail.h

RAIL_IEEE802154_SupportsEMultipurposeFrames

```
bool RAIL_IEEE802154_SupportsEMultipurposeFrames (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports IEEE 802.15.4E-2012 Multipurpose frame reception.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if Multipurpose frame reception is supported; false otherwise.

Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_E_MULTIPURPOSE_FRAMES](#).

Definition at line 7762 of file common/rail.h

RAIL_IEEE802154_SupportsESubsetGB868

```
bool RAIL_IEEE802154_SupportsESubsetGB868 (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the IEEE 802.15.4E-2012 feature subset needed for Zigbee R22 GB868.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if 802.15.4E GB868 subset is supported; false otherwise.

Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868](#).

Definition at line 7774 of file common/rail.h

RAIL_IEEE802154_SupportsG4ByteCrc

```
bool RAIL_IEEE802154_SupportsG4ByteCrc (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports IEEE 802.15.4G-2012 reception and transmission of frames with 4-byte CRC.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if 802.15.4G 4-byte CRC is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_G_4BYTE_CRC](#).

Definition at line 7786 of file `common/rail.h`

RAIL_IEEE802154_SupportsGDynFec

```
bool RAIL_IEEE802154_SupportsGDynFec (RAIL\_Handle\_t railHandle)
```

Indicate whether this chip supports IEEE 802.15.4G dynamic FEC.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if dynamic FEC is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_G_DYNFEC](#).

Definition at line 7797 of file `common/rail.h`

RAIL_SupportsProtocolWiSUN

```
bool RAIL_SupportsProtocolWiSUN (RAIL\_Handle\_t railHandle)
```

Indicate whether this chip supports Wi-SUN.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if Wi-SUN is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_PROTOCOL_WI_SUN](#).

Definition at line 7808 of file `common/rail.h`

RAIL_IEEE802154_SupportsGModeSwitch

```
bool RAIL_IEEE802154_SupportsGModeSwitch (RAIL\_Handle\_t railHandle)
```

Indicate whether this chip supports Wi-SUN mode switching.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

true if Wi-SUN mode switching is supported; false otherwise.

Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_G_MODESWITCH](#).

Definition at line 7819 of file common/rail.h

RAIL_IEEE802154_SupportsGSubsetGB868

bool RAIL_IEEE802154_SupportsGSubsetGB868 ([RAIL_Handle_t](#) railHandle)

Indicate whether this chip supports IEEE 802.15.4G-2012 feature subset needed for Zigbee R22 GB868.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if 802.15.4G GB868 subset is supported; false otherwise.

Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868](#).

Definition at line 7831 of file common/rail.h

RAIL_IEEE802154_SupportsGUnwhitenedRx

bool RAIL_IEEE802154_SupportsGUnwhitenedRx ([RAIL_Handle_t](#) railHandle)

Indicate whether this chip supports IEEE 802.15.4G-2012 reception of unwhitened frames.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if 802.15.4G unwhitened frame reception is supported; false otherwise.

Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_RX](#).

Definition at line 7844 of file common/rail.h

RAIL_IEEE802154_SupportsGUnwhitenedTx

bool RAIL_IEEE802154_SupportsGUnwhitenedTx ([RAIL_Handle_t](#) railHandle)

Indicate whether this chip supports IEEE 802.15.4G-2012 transmission of unwhitened frames.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if 802.15.4G unwhitened frame transmit is supported; false otherwise.

Runtime refinement of compile-time [RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_TX](#).

Definition at line 7857 of file common/rail.h

RAIL_WMBUS_SupportsSimultaneousTCRx

```
bool RAIL_WMBUS_SupportsSimultaneousTCRx (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports WMBUS simultaneous M2O RX of T and C modes.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if the WMBUS simultaneous M2O RX of T and C modes is supported; false otherwise.

Runtime refinement of compile-time [RAIL_WMBUS_SUPPORTS_SIMULTANEOUS_T_C_RX](#).

Definition at line 7867 of file common/rail.h

RAIL_SupportsProtocolZWave

```
bool RAIL_SupportsProtocolZWave (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the Z-Wave protocol.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if the Z-Wave protocol is supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_PROTOCOL_ZWAVE](#).

Definition at line 7877 of file common/rail.h

RAIL_ZWAVE_SupportsConcPhy

```
bool RAIL_ZWAVE_SupportsConcPhy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the Z-Wave concurrent PHY.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if the Z-Wave concurrent PHY is supported; false otherwise.

Runtime refinement of compile-time [RAIL_ZWAVE_SUPPORTS_CONC_PHY](#).

Definition at line 7887 of file common/rail.h

RAIL_ZWAVE_SupportsEnergyDetectPhy

bool RAIL_ZWAVE_SupportsEnergyDetectPhy ([RAIL_Handle_t](#) railHandle)

Indicate whether this chip supports the Z-Wave energy detect PHY.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if the Z-Wave energy detect PHY is supported; false otherwise.

Runtime refinement of compile-time [RAIL_ZWAVE_SUPPORTS_ED_PHY](#).

Definition at line 7897 of file `common/rail.h`

RAIL_ZWAVE_SupportsRegionPti

bool RAIL_ZWAVE_SupportsRegionPti ([RAIL_Handle_t](#) railHandle)

Indicate whether this chip supports Z-Wave Region in PTI.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if ZWAVE Region in PTI is supported; false otherwise.

Runtime refinement of compile-time [RAIL_ZWAVE_SUPPORTS_REGION_PTI](#).

Definition at line 7907 of file `common/rail.h`

RAIL_IEEE802154_SupportsSignalIdentifier

bool RAIL_IEEE802154_SupportsSignalIdentifier ([RAIL_Handle_t](#) railHandle)

Indicate whether this chip supports IEEE 802.15.4 signal identifier.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if signal identifier is supported; false otherwise.

Definition at line 7915 of file `common/rail.h`

RAIL_SupportsFastRx2Rx

bool RAIL_SupportsFastRx2Rx ([RAIL_Handle_t](#) railHandle)

Indicate whether this chip supports fast RX-to-RX.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if fast RX-to-RX is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_FAST_RX2RX](#).

Definition at line 7925 of file common/rail.h

RAIL_SupportsCollisionDetection

```
bool RAIL_SupportsCollisionDetection (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports collision detection.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if collision detection is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_COLLISION_DETECTION](#).

Definition at line 7935 of file common/rail.h

RAIL_SupportsProtocolSidewalk

```
bool RAIL_SupportsProtocolSidewalk (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports Sidewalk protocol.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if Sidewalk protocol is supported; false otherwise.
- Runtime refinement of compile-time [RAIL_SUPPORTS_PROTOCOL_SIDEWALK](#).

Definition at line 7945 of file common/rail.h

RAIL_SupportsTrustZoneSecurePeripherals

```
bool RAIL_SupportsTrustZoneSecurePeripherals (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports TrustZone secure configuration of peripherals used by RAIL.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if secure mode is supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_TRUSTZONE_SECURE_PERIPHERALS](#).

Definition at line 7956 of file `common/rail.h`

RAIL_SupportsPrsLnaBypass

```
bool RAIL_SupportsPrsLnaBypass (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports automatic PRS LNA bypass for external FEM.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if automatic PRS LNA bypass is supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_PRS_LNA_BYPASS](#).

Definition at line 7967 of file `common/rail.h`

RAIL_SupportsProtocolBTC

```
bool RAIL_SupportsProtocolBTC (RAIL_Handle_t railHandle)
```

Indicate whether RAIL supports the BTC protocol on this chip.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- true if BTC is supported; false otherwise.

Runtime refinement of compile-time [RAIL_SUPPORTS_PROTOCOL_BTC](#).

Definition at line 7977 of file `common/rail.h`

Front End Module Voltage Detection

Front End Module Voltage Detection

APIs for configuring and detecting a voltage from a Front End Module (VDET).

The rail_util_vdet plugin lets customers measure an output voltage from their attached Front End Module at a specific time in a Tx packet. This information allows optimizations to power configuration, and monitoring FEM performance.

Note

- VDET is only supported with EFR32xG25 devices.

Modules

[RAIL_VdetConfig_t](#)

Enumerations

```
enum RAIL\_Vdet\_Mode\_t {
    RAIL_VDET_MODE_DISABLED = 0u
    RAIL_VDET_MODE_AUTOMATIC = 1u
    RAIL_VDET_MODE_IMMEDIATE = 2u
    RAIL_VDET_MODE_COUNT
}
VDET Modes.

enum RAIL\_Vdet\_Resolution\_t {
    RAIL_VDET_RESOLUTION_10_BIT = 0u
    RAIL_VDET_RESOLUTION_11_BIT = 1u
    RAIL_VDET_RESOLUTION_12_BIT = 2u
    RAIL_VDET_RESOLUTION_COUNT
}
VDET Resolution for the Aux ADC.

enum RAIL\_Vdet\_Status\_t {
    RAIL_VDET_STATUS_IDLE = 0u
    RAIL_VDET_STATUS_START = 1u
    RAIL_VDET_STATUS_10_BIT_DONE = 2u
    RAIL_VDET_STATUS_11_BIT_DONE = 3u
    RAIL_VDET_STATUS_12_BIT_DONE = 4u
    RAIL_VDET_STATUS_BLOCKED = 5u
    RAIL_VDET_STATUS_ERROR = 6u
    RAIL_VDET_STATUS_COUNT
}
VDET Status.
```

Functions

RAIL_Status_t	RAIL_ConfigVdet (RAIL_Handle_t genericRailHandle, const RAIL_VdetConfig_t *config) Configure the VDET plugin.
RAIL_Status_t	RAIL_GetVdetConfig (RAIL_Handle_t genericRailHandle, RAIL_VdetConfig_t *config) Get the VDET plugin configuration.

RAIL_Status_t	RAIL_EnableVdet (RAIL_Handle_t genericRailHandle, bool enable) Enable the VDET plugin.
bool	RAIL_IsVdetEnabled (RAIL_Handle_t genericRailHandle) Check if the VDET plugin is enabled.
RAIL_Status_t	RAIL_GetVdet (RAIL_Handle_t genericRailHandle, uint32_t *pVdetMv) Get VDET measurement.

Macros

#define	RAIL_VDET_MODE_ENUM_NAMES undefined A macro that is string versions of the calibration enums.
#define	RAIL_VDET_RESOLUTION_ENUM_NAMES undefined A macro that is string versions of the resolution enums.
#define	RAIL_VDET_STATUS_ENUM_NAMES undefined A macro that is string versions of the status enums.

Enumeration Documentation

RAIL_Vdet_Mode_t

RAIL_Vdet_Mode_t

VDET Modes.

The VDET Mode is passed to [RAIL_ConfigVdet\(\)](#) via [RAIL_VdetConfig_t](#). The [Front End Module Voltage Detection \(VDET\) Utility](#) component allows customers to measure their Front End Module performance at specified points in the Transmit packet.

Enumerator	
RAIL_VDET_MODE_DISABLED	VDET is completely disabled.
RAIL_VDET_MODE_AUTOMATIC	AUTOMATIC causes VDET measurements to be taken every Tx packet at the specified time.
RAIL_VDET_MODE_IMMEDIATE	IMMEDIATE causes an immediate VDET measurement.
RAIL_VDET_MODE_COUNT	A count of the choices in this enumeration.

Definition at line 5637 of file `common/rail_types.h`

RAIL_Vdet_Resolution_t

RAIL_Vdet_Resolution_t

VDET Resolution for the Aux ADC.

The VDET Resolution is passed to [RAIL_ConfigVdet\(\)](#) via [RAIL_VdetConfig_t](#). Shows available resolution options.

Enumerator	
RAIL_VDET_RESOLUTION_10_BIT	~10 bit resolution.
RAIL_VDET_RESOLUTION_11_BIT	~11 bit resolution.
RAIL_VDET_RESOLUTION_12_BIT	~12 bit resolution.
RAIL_VDET_RESOLUTION_COUNT	A count of the choices in this enumeration.

Definition at line 5673 of file common/rail_types.h

RAIL_Vdet_Status_t

RAIL_Vdet_Status_t

VDET Status.

The VDET Status for internal debug. Shows states.

Enumerator	
RAIL_VDET_STATUS_IDLE	IDLE - Waiting for next command/measurement.
RAIL_VDET_STATUS_START	START of the VDET measurement activity.
RAIL_VDET_STATUS_10_BIT_DONE	Completion of a 10 bit measurement.
RAIL_VDET_STATUS_11_BIT_DONE	Completion of a 11 bit measurement.
RAIL_VDET_STATUS_12_BIT_DONE	Completion of a 12 bit measurement.
RAIL_VDET_STATUS_BLOCKED	Conflict with another AuxADC user.
RAIL_VDET_STATUS_ERROR	An error has occurred.
RAIL_VDET_STATUS_COUNT	A count of the choices in this enumeration.

Definition at line 5710 of file common/rail_types.h

Function Documentation

RAIL_ConfigVdet

RAIL_Status_t RAIL_ConfigVdet (RAIL_Handle_t genericRailHandle, const RAIL_VdetConfig_t * config)

Configure the VDET plugin.

Parameters

[in]	genericRailHandle	A radio-generic RAIL handle.
[in]	config	A pointer to a RAIL_VdetConfig_t struct that contains configuration data for the VDET.

Returns

- RAIL_Status_t

Return values

- RAIL_STATUS_NO_ERROR: - All went well
- RAIL_STATUS_INVALID_STATE: - VDET is enabled. Must be disabled first.
- RAIL_STATUS_INVALID_PARAMETER: - mode/resolution/delayUs out-of-bounds.

Warnings

- As this function relies on GPIO access and RAIL is meant to run in TrustZone non-secure world, it is not supported if GPIO is configured as secure peripheral and it will return RAIL_STATUS_INVALID_CALL.

Definition at line 6401 of file common/rail.h

RAIL_GetVdetConfig

```
RAIL_Status_t RAIL_GetVdetConfig (RAIL_Handle_t genericRailHandle, RAIL_VdetConfig_t * config)
```

Get the VDET plugin configuration.

Parameters

[in]	genericRailHandle	A radio-generic RAIL handle.
[out]	config	A pointer to a RAIL_VdetConfig_t struct that will return configuration data for the VDET.

Returns

- [RAIL_Status_t](#)

Return values

- RAIL_STATUS_NO_ERROR: - All went well.

Definition at line 6413 of file `common/rail.h`

RAIL_EnableVdet

```
RAIL_Status_t RAIL_EnableVdet (RAIL_Handle_t genericRailHandle, bool enable)
```

Enable the VDET plugin.

Parameters

[in]	genericRailHandle	A RAIL instance handle.
[in]	enable	Enable or disable RAIL_VDET_MODE_AUTOMATIC mode captures. Begins measurement if in RAIL_VDET_MODE_IMMEDIATE .

Returns

- [RAIL_Status_t](#)

Return values

- RAIL_STATUS_NO_ERROR: - All went well, VDET is enabled or disabled.
- RAIL_STATUS_INVALID_STATE: - VDET has not been configured or VDET was not idle. VDET is disabled.

Warnings

- As this function relies on HFXO access and RAIL is meant to run in TrustZone non-secure world, it is not supported if HFXO is configured as secure peripheral and it will return [RAIL_STATUS_INVALID_CALL](#).

Definition at line 6431 of file `common/rail.h`

RAIL_IsVdetEnabled

```
bool RAIL_IsVdetEnabled (RAIL_Handle_t genericRailHandle)
```

Check if the VDET plugin is enabled.

Parameters

[in]	genericRailHandle	A RAIL instance handle.

Returns

- bool

Definition at line 6440 of file common/rail.h

RAIL_GetVdet

RAIL_Status_t RAIL_GetVdet ([RAIL_Handle_t](#) genericRailHandle, uint32_t * pVdetMv)

Get VDET measurement.

Parameters

[in]	genericRailHandle	A RAIL instance handle.
[out]	pVdetMv	A pointer to a uint32_t for returning the measurement.

A processed measurement in millivolts.

Returns

- [RAIL_Status_t](#)

Return values

- RAIL_STATUS_NO_ERROR: - All went well, value in *pVdetMv is valid.
- RAIL_STATUS_INVALID_CALL: - VDET is in progress, wait until VDET capture is complete and try again.
- RAIL_STATUS_INVALID_STATE: - VDET is not enabled or valid. Wait until next packet and try reading again.
- RAIL_STATUS_INVALID_PARAMETER: - In
- RAIL_STATUS_SUSPENDED: - Blocked by AuxADC contention. Wait until next packet and try reading again.

Warnings

- As this function relies on HFXO access and RAIL is meant to run in TrustZone non-secure world, it is not supported if HFXO is configured as secure peripheral and it will return [RAIL_STATUS_INVALID_CALL](#).

Definition at line 6463 of file common/rail.h

RAIL_VdetConfig_t

Configuration information for FEM Voltage Detection plugin.

A structure of type [RAIL_VdetConfig_t](#) is passed to [RAIL_ConfigVdet\(\)](#).

Public Attributes

RAIL_Vdet_Mode_t	mode	Mode for the VDET.
RAIL_Vdet_Resolution_t	resolution	Resolution to use for the capture.
uint32_t	delayUs	Delay in microseconds for the capture from Tx Start in RAIL_VDET_MODE_AUTOMATIC .

Public Attribute Documentation

mode

`RAIL_Vdet_Mode_t RAIL_VdetConfig_t::mode`

Mode for the VDET.

Definition at line 5763 of file `common/rail_types.h`

resolution

`RAIL_Vdet_Resolution_t RAIL_VdetConfig_t::resolution`

Resolution to use for the capture.

Definition at line 5765 of file `common/rail_types.h`

delayUs

`uint32_t RAIL_VdetConfig_t::delayUs`

Delay in microseconds for the capture from Tx Start in [RAIL_VDET_MODE_AUTOMATIC](#).

Minimum 5 us, maximum 100000 us.

Definition at line 5767 of file `common/rail_types.h`

General

General

Basic APIs to set up and interact with the RAIL library.

Modules

- [RAIL_Version_t](#)
- [RAILSched_Config_t](#)
- [RAIL_Config_t](#)
- [EFR32xG2x](#)
- [SIxx3xx](#)

Typedefs

typedef void *	RAIL_Handle_t A radio-generic handle (e.g., RAIL_EFR32_HANDLE), or a real RAIL instance handle as returned from RAIL_Init() .
typedef sl_status_t	RAIL_Status_t A status returned by many RAIL API calls indicating their success or failure.
typedef void(*)	RAIL_InitCompleteCallbackPtr_t (RAIL_Handle_t railHandle) A pointer to an initialization complete callback function.
typedef uint8_t	RAIL_StateBuffer_t [1] Provided for backwards compatibility.

Functions

RAIL_Status_t	RAIL_GetVersion (RAIL_Version_t *version, bool verbose) Get the version information for the compiled RAIL library.
RAIL_Status_t	RAIL_AddStateBuffer3 (RAIL_Handle_t genericRailHandle) Add a 3rd multiprotocol internal state buffer for use by RAIL_Init() .
RAIL_Status_t	RAIL_AddStateBuffer4 (RAIL_Handle_t genericRailHandle) Add a 4th multiprotocol internal state buffer for use by RAIL_Init() .
RAIL_Status_t	RAIL_UseDma (uint8_t channel) Allocate a DMA channel for RAIL to work with.
RAIL_Status_t	RAIL_CopyDeviceInfo (RAIL_Handle_t genericRailHandle) Reads out device specific data that may be needed by RAIL and populates appropriate data structures in the library.
RAIL_Handle_t	RAIL_Init (const RAIL_Config_t *railCfg, RAIL_InitCompleteCallbackPtr_t cb) Initialize RAIL.
bool	RAIL_IsInitialized (void) Get RAIL initialization status.

uint16_t

[RAIL_GetRadioEntropy](#)(RAIL_Handle_t railHandle, uint8_t *buffer, uint16_t bytes)
Collect entropy from the radio if available.

Macros

- #define RAIL_EFR32_HANDLE ((RAIL_Handle_t)0xFFFFFFFFUL)
A placeholder for a radio-generic RAIL handle.
- #define RAIL_STATUS_NO_ERROR SL_STATUS_OK
RAIL function reports no error.
- #define RAIL_STATUS_INVALID_PARAMETER SL_STATUS_INVALID_PARAMETER
Call to RAIL function threw an error because of an invalid parameter.
- #define RAIL_STATUS_INVALID_STATE SL_STATUS_INVALID_STATE
Call to RAIL function threw an error because it was called during an invalid radio state.
- #define RAIL_STATUS_INVALID_CALL SL_STATUS_NOT_AVAILABLE
RAIL function is called in an invalid order.
- #define RAIL_STATUS_SUSPENDED SL_STATUS_IN_PROGRESS
RAIL function did not finish in the allotted time.
- #define RAIL_STATUS_SCHED_ERROR SL_STATUS_ABORT
RAIL function could not be scheduled by the Radio scheduler.
- #define RAIL_DMA_INVALID (0xFFU)
A value to signal that RAIL should not use DMA.

Typedef Documentation

RAIL_Handle_t

RAIL_Handle_t

A radio-generic handle (e.g., [RAIL_EFR32_HANDLE](#)), or a real RAIL instance handle as returned from [RAIL_Init\(\)](#).

Generic handles should be used for certain RAIL APIs that are called prior to RAIL initialization. However, once RAIL has been initialized, the real handle returned by [RAIL_Init\(\)](#) should be used instead.

Definition at line 109 of file `common/rail_types.h`

RAIL_Status_t

RAIL_Status_t

A status returned by many RAIL API calls indicating their success or failure.

It is a subset of `sl_status_t`.

Definition at line 126 of file `common/rail_types.h`

RAIL_InitCompleteCallbackPtr_t


```
typedef void(* RAIL_InitCompleteCallbackPtr_t) (RAIL_Handle_t railHandle) (RAIL_Handle_t railHandle)
```

A pointer to an initialization complete callback function.

Parameters

[in]	railHandle	The initialized RAIL instance handle.

Definition at line 157 of file common/rail_types.h

RAIL_StateBuffer_t

```
RAIL_StateBuffer_t [1]
```

Provided for backwards compatibility.

Definition at line 175 of file common/rail_types.h

Function Documentation

RAIL_GetVersion

```
RAIL_Status_t RAIL_GetVersion (RAIL_Version_t * version, bool verbose)
```

Get the version information for the compiled RAIL library.

Parameters

[out]	version	A non-NULL pointer to RAIL_Version_t structure to populate with version information.
[in]	verbose	Populate RAIL_Version_t struct with verbose information.

Returns

- Status code indicating success of the function call.
- The version information contains a major version number, a minor version number, and a rev (revision) number.

Definition at line 85 of file common/rail.h

RAIL_AddStateBuffer3

```
RAIL_Status_t RAIL_AddStateBuffer3 (RAIL_Handle_t genericRailHandle)
```

Add a 3rd multiprotocol internal state buffer for use by [RAIL_Init\(\)](#).

Parameters

[in]	genericRailHandle	A generic RAIL instance handle.

Returns

- Status code indicating success of the function call. An error is returned if the 3rd state buffer was previously added or this isn't the RAIL multiprotocol library.

Definition at line 154 of file common/rail.h

RAIL_AddStateBuffer4

RAIL_Status_t RAIL_AddStateBuffer4 (RAIL_Handle_t genericRailHandle)

Add a 4th multiprotocol internal state buffer for use by RAIL_Init().

Parameters

[in]	genericRailHandle	A generic RAIL instance handle.

Returns

- Status code indicating success of the function call. An error is returned if the 4th state buffer was previously added. or this isn't the RAIL multiprotocol library.

Definition at line 164 of file common/rail.h

RAIL_UseDma

RAIL_Status_t RAIL_UseDma (uint8_t channel)

Allocate a DMA channel for RAIL to work with.

Parameters

[in]	channel	The DMA channel to use when copying memory. If a value of RAIL_DMA_INVALID is passed, RAIL will stop using any DMA channel.

Returns

- Status code indicating success of the function call.

To use this API, the application must initialize the DMA engine on the chip and allocate a DMA channel. This channel will be used periodically to copy memory more efficiently. Call this function before RAIL_Init() to have the most benefit. If the application needs to take back control of the DMA channel that RAIL is using, this API may be called with a channel of RAIL_DMA_INVALID to tell RAIL to stop using DMA.

Warnings

- To allocate and use a DMA channel for RAIL to work with when TrustZone is enabled and LDMA is configured as secure peripheral, the secure application must initialize the DMA engine and call this API. The non-secure application must provide a non-NULL RAIL_TZ_Config_t::radioPerformM2mLdmaCallback to RAIL_TZ_InitNonSecure(). To take back control of the DMA channel when TrustZone is enabled and LDMA is configured as secure peripheral, the secure application must call this API with a channel of RAIL_DMA_INVALID. The non-secure application must provide a NULL RAIL_TZ_Config_t::radioPerformM2mLdmaCallback to RAIL_TZ_InitNonSecure().

Definition at line 192 of file common/rail.h

RAIL_CopyDeviceInfo

RAIL_Status_t RAIL_CopyDeviceInfo (RAIL_Handle_t genericRailHandle)

Reads out device specific data that may be needed by RAIL and populates appropriate data structures in the library.

Parameters

[in]	genericRailHandle	A generic RAIL instance handle.

Returns

- Status code indicating success of the function call.

Note

- This function must be called before calling [RAIL_Init\(\)](#) on any platforms that require this data and should not be called inside a critical section. This function does nothing on EFR32 Series 2 devices.

Definition at line 314 of file `common/rail.h`

RAIL_Init

```
RAIL_Handle_t RAIL_Init (const RAIL\_Config\_t * railCfg, RAIL\_InitCompleteCallbackPtr\_t cb)
```

Initialize RAIL.

Parameters

[in]	railCfg	The configuration for setting up the protocol.
[in]	cb	A callback that notifies the application when the radio is finished initializing and is ready for further configuration. This callback is useful for potential transceiver products that require a power up sequence before further configuration is available. After the callback fires, the radio is ready for additional configuration before transmit and receive operations.

Returns

- Handle for initialized rail instance or NULL if an invalid value was passed in the railCfg.

Note

- Call this function only once per protocol. If called again, it will do nothing and return NULL. [RAIL_CopyDeviceInfo\(\)](#) should be called once before calling this function for Silicon Labs Series 3 devices.
- The first call to [RAIL_Init\(\)](#) implicitly enables PTI, but it won't take effect unless or until [RAIL_ConfigPti\(\)](#) has been called with a mode other than the default [RAIL_PTI_MODE_DISABLED](#).

Definition at line 340 of file `common/rail.h`

RAIL_IsInitialized

```
bool RAIL_IsInitialized (void )
```

Get RAIL initialization status.

Parameters

N/A		

Returns

- true if the radio has finished initializing and false otherwise.

RAIL APIs, e.g., [RAIL_GetTime\(\)](#), which work only if [RAIL_Init\(\)](#) has been called, can use [RAIL_IsInitialized\(\)](#) to determine whether RAIL has been initialized or not.

Definition at line 352 of file `common/rail.h`

RAIL_GetRadioEntropy

```
uint16_t RAIL_GetRadioEntropy (RAIL_Handle_t railHandle, uint8_t * buffer, uint16_t bytes)
```

Collect entropy from the radio if available.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	buffer	A non-NULL pointer to the buffer to write the collected entropy.
[in]	bytes	The number of bytes to fill in the input buffer.

Returns

- The number of bytes of entropy collected. For radios that don't support entropy collection, the function returns 0. Values less than the requested amount may also be returned on platforms that use entropy pools to collect random data periodically.

Attempts to fill the provided buffer with the requested number of bytes of entropy. If the requested number of bytes can't be provided, as many bytes as possible will be filled and returned. For radios that do not support this function, 0 bytes are always returned. For information about the specific mechanism for gathering entropy, see documentation for the chip family.

Definition at line 374 of file common/rail.h

RAIL_Version_t

Contains RAIL Library Version Information.

It is filled in by [RAIL_GetVersion\(\)](#).

Public Attributes

uint32_t	hash	Git hash.
uint8_t	major	Major number.
uint8_t	minor	Minor number.
uint8_t	rev	Revision number.
uint8_t	build	Build number.
uint8_t	flags	Build flags.
bool	multiprotocol	Boolean to indicate whether this is a multiprotocol library or not.

Public Attribute Documentation

hash

uint32_t RAIL_Version_t::hash

Git hash.

Definition at line 85 of file `common/rail_types.h`

major

uint8_t RAIL_Version_t::major

Major number.

Definition at line 87 of file `common/rail_types.h`

minor

uint8_t RAIL_Version_t::minor

Minor number.

Definition at line 89 of file common/rail_types.h

rev

```
uint8_t RAIL_Version_t::rev
```

Revision number.

Definition at line 91 of file common/rail_types.h

build

```
uint8_t RAIL_Version_t::build
```

Build number.

Definition at line 93 of file common/rail_types.h

flags

```
uint8_t RAIL_Version_t::flags
```

Build flags.

Definition at line 95 of file common/rail_types.h

multiprotocol

```
bool RAIL_Version_t::multiprotocol
```

Boolean to indicate whether this is a multiprotocol library or not.

Definition at line 97 of file common/rail_types.h

RAILSched_Config_t

Provided for backwards compatibility.

Public Attributes

uint8_t

buffer

Dummy buffer no longer used.

Public Attribute Documentation

buffer

uint8_t RAILSched_Config_t::buffer[1]

Dummy buffer no longer used.

Definition at line 168 of file common/rail_types.h

RAIL_Config_t

RAIL configuration structure.

Public Attributes

<code>void(*</code>	<code>eventsCallback</code>	A pointer to a function, which is called whenever a RAIL event occurs.
<code>void *</code>	<code>protocol</code>	Provided for backwards compatibility.
<code>RAILSched_Config_t *</code>	<code>scheduler</code>	Provided for backwards compatibility.
<code>RAIL_StateBuffer_t</code>	<code>buffer</code>	Provided for backwards compatibility.

Public Attribute Documentation

eventsCallback

```
void(* RAIL_Config_t::eventsCallback) (RAIL_Handle_t railHandle, RAIL_Events_t events)
```

A pointer to a function, which is called whenever a RAIL event occurs.

See the [RAIL_Events_t](#) documentation for the list of RAIL events.

Definition at line 1687 of file common/rail_types.h

protocol

```
void* RAIL_Config_t::protocol
```

Provided for backwards compatibility.

Ignored.

Definition at line 1691 of file common/rail_types.h

scheduler

```
RAILSched_Config_t* RAIL_Config_t::scheduler
```

Provided for backwards compatibility.

Ignored.

Definition at line 1695 of file common/rail_types.h

buffer

RAIL_StateBuffer_t RAIL_Config_t::buffer

Provided for backwards compatibility.

Ignored.

Definition at line 1699 of file common/rail_types.h

EFR32xG2x

EFR32xG2x

Types specific to the EFR32xG2x for general configuration.

Slxx3xx

Slxx3xx

Types specific to the Slxx3xx for general configuration.

Macros

#define	RAIL_S3LPW_HANDLE RAIL_EFR32_HANDLE
	Synonym of RAIL_EFR32_HANDLE for Series 3.

Multiprotocol

Multiprotocol

Multiprotocol scheduler APIs to support multiple time-sliced PHYs.

Modules

[RAIL_SchedulerInfo_t](#)

[EFR32xG2x](#)

[Slxx3xx](#)

Enumerations

```
enum RAIL_SchedulerStatus_t {
    RAIL_SCHEDULER_STATUS_NO_ERROR = (0U << 0)
    RAIL_SCHEDULER_STATUS_UNSUPPORTED = (1U << 0)
    RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED = (2U << 0)
    RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL = (3U << 0)
    RAIL_SCHEDULER_STATUS_TASK_FAIL = (4U << 0)
    RAIL_SCHEDULER_STATUS_INTERNAL_ERROR = (5U << 0)
    RAIL_SCHEDULER_TASK_EMPTY = (0U << 4)
    RAIL_SCHEDULER_TASK_SCHEDULED_RX = (1U << 4)
    RAIL_SCHEDULER_TASK_SCHEDULED_TX = (2U << 4)
    RAIL_SCHEDULER_TASK_SINGLE_TX = (3U << 4)
    RAIL_SCHEDULER_TASK_SINGLE_CCA_CSMA_TX = (4U << 4)
    RAIL_SCHEDULER_TASK_SINGLE_CCA_LBT_TX = (5U << 4)
    RAIL_SCHEDULER_TASK_SCHEDULED_CCA_CSMA_TX = (6U << 4)
    RAIL_SCHEDULER_TASK_SCHEDULED_CCA_LBT_TX = (7U << 4)
    RAIL_SCHEDULER_TASK_TX_STREAM = (8U << 4)
    RAIL_SCHEDULER_TASK_AVERAGE_RSSI = (9U << 4)
    RAIL_SCHEDULER_STATUS_SCHEDULED_TX_FAIL = (RAIL_SCHEDULER_TASK_SCHEDULED_TX |
    RAIL_SCHEDULER_STATUS_TASK_FAIL)
    RAIL_SCHEDULER_STATUS_SINGLE_TX_FAIL = (RAIL_SCHEDULER_TASK_SINGLE_TX |
    RAIL_SCHEDULER_STATUS_TASK_FAIL)
    RAIL_SCHEDULER_STATUS_CCA_CSMA_TX_FAIL = (RAIL_SCHEDULER_TASK_SINGLE_CCA_CSMA_TX |
    RAIL_SCHEDULER_STATUS_TASK_FAIL)
    RAIL_SCHEDULER_STATUS_CCA_LBT_TX_FAIL = (RAIL_SCHEDULER_TASK_SINGLE_CCA_LBT_TX |
    RAIL_SCHEDULER_STATUS_TASK_FAIL)
    RAIL_SCHEDULER_STATUS_SCHEDULED_RX_FAIL = (RAIL_SCHEDULER_TASK_SCHEDULED_RX |
    RAIL_SCHEDULER_STATUS_TASK_FAIL)
    RAIL_SCHEDULER_STATUS_TX_STREAM_FAIL = (RAIL_SCHEDULER_TASK_TX_STREAM |
    RAIL_SCHEDULER_STATUS_TASK_FAIL)
    RAIL_SCHEDULER_STATUS_AVERAGE_RSSI_FAIL = (RAIL_SCHEDULER_TASK_AVERAGE_RSSI |
    RAIL_SCHEDULER_STATUS_TASK_FAIL)
    RAIL_SCHEDULER_SCHEDULED_RX_INTERNAL_ERROR = (RAIL_SCHEDULER_TASK_SCHEDULED_RX |
    RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
    RAIL_SCHEDULER_SCHEDULED_RX_SCHEDULING_ERROR = (RAIL_SCHEDULER_TASK_SCHEDULED_RX |
    RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
    RAIL_SCHEDULER_SCHEDULED_RX_INTERRUPTED = (RAIL_SCHEDULER_TASK_SCHEDULED_RX |
    RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
    RAIL_SCHEDULER_SCHEDULED_TX_INTERNAL_ERROR = (RAIL_SCHEDULER_TASK_SCHEDULED_TX |
    RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
    RAIL_SCHEDULER_SCHEDULED_TX_SCHEDULING_ERROR = (RAIL_SCHEDULER_TASK_SCHEDULED_TX |
    RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
    RAIL_SCHEDULER_SCHEDULED_TX_INTERRUPTED = (RAIL_SCHEDULER_TASK_SCHEDULED_TX |
    RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
    RAIL_SCHEDULER_SINGLE_TX_INTERNAL_ERROR = (RAIL_SCHEDULER_TASK_SINGLE_TX |
    RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
    RAIL_SCHEDULER_SINGLE_TX_SCHEDULING_ERROR = (RAIL_SCHEDULER_TASK_SINGLE_TX |
    RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
    RAIL_SCHEDULER_SINGLE_TX_INTERRUPTED = (RAIL_SCHEDULER_TASK_SINGLE_TX |
    RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
    RAIL_SCHEDULER_SINGLE_CCA_CSMA_TX_INTERNAL_ERROR =
    (RAIL_SCHEDULER_TASK_SINGLE_CCA_CSMA_TX | RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
    RAIL_SCHEDULER_SINGLE_CCA_CSMA_TX_SCHEDULING_ERROR =
    (RAIL_SCHEDULER_TASK_SINGLE_CCA_CSMA_TX | RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
    RAIL_SCHEDULER_SINGLE_CCA_CSMA_TX_INTERRUPTED =
    (RAIL_SCHEDULER_TASK_SINGLE_CCA_CSMA_TX | RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
    RAIL_SCHEDULER_SINGLE_CCA_LBT_TX_INTERNAL_ERROR =
    (RAIL_SCHEDULER_TASK_SINGLE_CCA_LBT_TX | RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
    RAIL_SCHEDULER_SINGLE_CCA_LBT_TX_SCHEDULING_ERROR =
    (RAIL_SCHEDULER_TASK_SINGLE_CCA_LBT_TX | RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
    RAIL_SCHEDULER_SINGLE_CCA_LBT_TX_INTERRUPTED = (RAIL_SCHEDULER_TASK_SINGLE_CCA_LBT_TX |
    RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
    RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_INTERNAL_ERROR =
    (RAIL_SCHEDULER_TASK_SCHEDULED_CCA_CSMA_TX | RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
    RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_FAIL =
    (RAIL_SCHEDULER_TASK_SCHEDULED_CCA_CSMA_TX | RAIL_SCHEDULER_STATUS_TASK_FAIL)
    RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_SCHEDULING_ERROR =
    (RAIL_SCHEDULER_TASK_SCHEDULED_CCA_CSMA_TX | RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
    RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_INTERRUPTED =
    (RAIL_SCHEDULER_TASK_SCHEDULED_CCA_CSMA_TX |
    RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
}
```

```

RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_INTERNAL_ERROR =
(RAIL_SCHEDULER_TASK_SCHEDULED_CCA_LBT_TX |
RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_FAIL =
(RAIL_SCHEDULER_TASK_SCHEDULED_CCA_LBT_TX |
RAIL_SCHEDULER_STATUS_TASK_FAIL)
RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_SCHEDULING_ERROR
= (RAIL_SCHEDULER_TASK_SCHEDULED_CCA_LBT_TX |
RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_INTERRUPTED =
(RAIL_SCHEDULER_TASK_SCHEDULED_CCA_LBT_TX |
RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
RAIL_SCHEDULER_TX_STREAM_INTERNAL_ERROR =
(RAIL_SCHEDULER_TASK_TX_STREAM |
RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
RAIL_SCHEDULER_TX_STREAM_SCHEDULING_ERROR =
(RAIL_SCHEDULER_TASK_TX_STREAM |
RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
RAIL_SCHEDULER_TX_STREAM_INTERRUPTED =
(RAIL_SCHEDULER_TASK_TX_STREAM |
RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
RAIL_SCHEDULER_AVERAGE_RSSI_INTERNAL_ERROR =
(RAIL_SCHEDULER_TASK_AVERAGE_RSSI |
RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
RAIL_SCHEDULER_AVERAGE_RSSI_SCHEDULING_ERROR =
(RAIL_SCHEDULER_TASK_AVERAGE_RSSI |
RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
RAIL_SCHEDULER_AVERAGE_RSSI_INTERRUPTED =
(RAIL_SCHEDULER_TASK_AVERAGE_RSSI |
RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
}

```

Multiprotocol scheduler status returned by [RAIL_GetSchedulerStatus\(\)](#).

```

enum    RAIL_TaskType_t {
        RAIL_TASK_TYPE_START_RX = 0
        RAIL_TASK_TYPE_OTHER = 1
    }

```

Multiprotocol radio operation task types, used with [RAIL_SetTaskPriority\(\)](#).

Functions

RAIL_Status_t	RAIL_YieldRadio (RAIL_Handle_t railHandle) Yield the radio to other configurations.
RAIL_SchedulerStatus_t	RAIL_GetSchedulerStatus (RAIL_Handle_t railHandle) Get the status of the RAIL scheduler.
RAIL_Status_t	RAIL_GetSchedulerStatusAlt (RAIL_Handle_t railHandle, RAIL_SchedulerStatus_t *pSchedulerStatus, RAIL_Status_t *pRailStatus) Get the status of the RAIL scheduler, specific to the radio operation, along with RAIL_Status_t returned by RAIL API invoked by the RAIL scheduler.
RAIL_Status_t	RAIL_SetTaskPriority (RAIL_Handle_t railHandle, uint8_t priority, RAIL_TaskType_t taskType) Change the priority of a specified task type in multiprotocol.
RAIL_Time_t	RAIL_GetTransitionTime (void) Get the time needed to switch between protocols.
void	RAIL_SetTransitionTime (RAIL_Time_t transitionTime) Set the time needed to switch between protocols.

Macros

```
#define RAIL_SCHEDULER_STATUS_MASK 0x0FU
Radio Scheduler Status mask within RAIL\_SchedulerStatus\_t values.

#define RAIL_SCHEDULER_STATUS_SHIFT 0
Radio Scheduler Status shift within RAIL\_SchedulerStatus\_t values.

#define RAIL_SCHEDULER_TASK_MASK 0xF0U
Radio Scheduler Task mask within RAIL\_SchedulerStatus\_t values.

#define RAIL_SCHEDULER_TASK_SHIFT 4
Radio Scheduler Task shift within RAIL\_SchedulerStatus\_t values.
```

Enumeration Documentation

RAIL_SchedulerStatus_t

RAIL_SchedulerStatus_t

Multiprotocol scheduler status returned by [RAIL_GetSchedulerStatus\(\)](#).

[Multiprotocol](#) scheduler status is a combination of the upper 4 bits which constitute the type of scheduler task and the lower 4 bits which constitute the type of scheduler error.

Enumerator	
RAIL_SCHEDULER_STATUS_NO_ERROR	Multiprotocol scheduler reports no error.
RAIL_SCHEDULER_STATUS_UNSUPPORTED	The scheduler is disabled or the requested scheduler operation is unsupported.
RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED	The scheduled task was started but was interrupted by a higher-priority event before it could be completed.
RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL	Scheduled task could not be scheduled given its priority and the other tasks running on the system.
RAIL_SCHEDULER_STATUS_TASK_FAIL	Calling the RAIL API associated with the Radio scheduler task returned an error code.
RAIL_SCHEDULER_STATUS_INTERNAL_ERROR	An internal error occurred in scheduler data structures, which should not happen and indicates a problem.
RAIL_SCHEDULER_TASK_EMPTY	Radio scheduler has no task.
RAIL_SCHEDULER_TASK_SCHEDULED_RX	Radio scheduler calls RAIL_ScheduleRx() .
RAIL_SCHEDULER_TASK_SCHEDULED_TX	Radio scheduler calls RAIL_StartScheduledTx() .
RAIL_SCHEDULER_TASK_SINGLE_TX	Radio scheduler calls RAIL_StartTx() .
RAIL_SCHEDULER_TASK_SINGLE_CCA_CSMA_TX	Radio scheduler calls RAIL_StartCcaCsmaTx() .
RAIL_SCHEDULER_TASK_SINGLE_CCA_LBT_TX	Radio scheduler calls RAIL_StartCcaLbtTx() .
RAIL_SCHEDULER_TASK_SCHEDULED_CCA_CSMA_TX	Radio scheduler calls RAIL_StartScheduledCcaCsmaTx() .
RAIL_SCHEDULER_TASK_SCHEDULED_CCA_LBT_TX	Radio scheduler calls RAIL_StartScheduledCcaLbtTx() .
RAIL_SCHEDULER_TASK_TX_STREAM	Radio scheduler calls RAIL_StartTxStream() .
RAIL_SCHEDULER_TASK_AVERAGE_RSSI	Radio scheduler calls RAIL_StartAverageRssi() .
RAIL_SCHEDULER_STATUS_SCHEDULED_TX_FAIL	RAIL_StartScheduledTx() returned error status.
RAIL_SCHEDULER_STATUS_SINGLE_TX_FAIL	RAIL_StartTx() returned error status.
RAIL_SCHEDULER_STATUS_CCA_CSMA_TX_FAIL	RAIL_StartCcaCsmaTx() returned error status.
RAIL_SCHEDULER_STATUS_CCA_LBT_TX_FAIL	RAIL_StartCcaLbtTx() returned error status.
RAIL_SCHEDULER_STATUS_SCHEDULED_RX_FAIL	RAIL_ScheduleRx() returned error status.

RAIL_SCHEDULER_STATUS_TX_STREAM_FAIL	RAIL_StartTxStream() returned error status.
RAIL_SCHEDULER_STATUS_AVERAGE_RSSI_FAIL	RAIL_StartAverageRssi() returned error status.
RAIL_SCHEDULER_SCHEDULED_RX_INTERNAL_ERROR	Multiprotocol scheduled receive function internal error.
RAIL_SCHEDULER_SCHEDULED_RX_SCHEDULING_ERROR	Multiprotocol scheduled receive scheduling error.
RAIL_SCHEDULER_SCHEDULED_RX_INTERRUPTED	RAIL_ScheduleRx() operation interrupted
RAIL_SCHEDULER_SCHEDULED_TX_INTERNAL_ERROR	Multiprotocol scheduled TX internal error.
RAIL_SCHEDULER_SCHEDULED_TX_SCHEDULING_ERROR	Multiprotocol scheduled TX scheduling error.
RAIL_SCHEDULER_SCHEDULED_TX_INTERRUPTED	RAIL_StartScheduledTx() operation interrupted.
RAIL_SCHEDULER_SINGLE_TX_INTERNAL_ERROR	Multiprotocol instantaneous TX internal error.
RAIL_SCHEDULER_SINGLE_TX_SCHEDULING_ERROR	Multiprotocol instantaneous TX scheduling error.
RAIL_SCHEDULER_SINGLE_TX_INTERRUPTED	RAIL_StartTx() operation interrupted.
RAIL_SCHEDULER_SINGLE_CCA_CSMA_TX_INTERNAL_ERROR	Multiprotocol single CSMA transmit function internal error.
RAIL_SCHEDULER_SINGLE_CCA_CSMA_TX_SCHEDULING_ERROR	Multiprotocol single CSMA transmit scheduling error.
RAIL_SCHEDULER_SINGLE_CCA_CSMA_TX_INTERRUPTED	RAIL_StartCcaCsmatx() operation interrupted.
RAIL_SCHEDULER_SINGLE_CCA_LBT_TX_INTERNAL_ERROR	Multiprotocol single LBT transmit function internal error.
RAIL_SCHEDULER_SINGLE_CCA_LBT_TX_SCHEDULING_ERROR	Multiprotocol single LBT transmit scheduling error.
RAIL_SCHEDULER_SINGLE_CCA_LBT_TX_INTERRUPTED	RAIL_StartCcaLbtTx() operation interrupted.
RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_INTERNAL_ERROR	Multiprotocol scheduled CSMA transmit function internal error.
RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_FAIL	RAIL_StartScheduledCcaCsmatx() returned error status.
RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_SCHEDULING_ERROR	Multiprotocol scheduled CSMA transmit scheduling error.
RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_INTERRUPTED	RAIL_StartScheduledCcaCsmatx() operation interrupted.
RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_INTERNAL_ERROR	Multiprotocol scheduled LBT transmit function internal error.
RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_FAIL	RAIL_StartScheduledCcaLbtTx() returned error status.
RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_SCHEDULING_ERROR	Multiprotocol scheduled LBT transmit scheduling error.
RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_INTERRUPTED	RAIL_StartScheduledCcaLbtTx() operation interrupted.
RAIL_SCHEDULER_TX_STREAM_INTERNAL_ERROR	Multiprotocol stream transmit function internal error.
RAIL_SCHEDULER_TX_STREAM_SCHEDULING_ERROR	Multiprotocol stream transmit scheduling error.
RAIL_SCHEDULER_TX_STREAM_INTERRUPTED	RAIL_StartTxStream() operation interrupted.
RAIL_SCHEDULER_AVERAGE_RSSI_INTERNAL_ERROR	Multiprotocol RSSI averaging function internal error.
RAIL_SCHEDULER_AVERAGE_RSSI_SCHEDULING_ERROR	Multiprotocol RSSI average scheduling error.
RAIL_SCHEDULER_AVERAGE_RSSI_INTERRUPTED	RAIL_StartAverageRssi() operation interrupted.

Definition at line 547 of file common/rail_types.h

RAIL_TaskType_t

RAIL_TaskType_t

Multiprotocol radio operation task types, used with [RAIL_SetTaskPriority\(\)](#).

Enumerator

RAIL_TASK_TYPE_START_RX	Indicate a task started using RAIL_StartRx() .
RAIL_TASK_TYPE_OTHER	Indicate a task started functions other than RAIL_StartRx() .

Definition at line 779 of file `common/railTypes.h`

Function Documentation

RAIL_YieldRadio

RAIL_Status_t RAIL_YieldRadio ([RAIL_Handle_t](#) railHandle)

Yield the radio to other configurations.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

This function is used to indicate that the previous transmit or scheduled receive operation has completed. It must be used in multiprotocol RAIL because the scheduler assumes that any transmit or receive operation that is started can go on indefinitely based on state transitions and your protocol. RAIL will not allow a lower priority tasks to run until this is called so it can negatively impact performance of those protocols if this is omitted or delayed. It is also possible to call the [RAIL_Idle\(\)](#) API to both terminate the operation and idle the radio. In single protocol RAIL this API does nothing, however, if RAIL Power Manager is initialized, calling [RAIL_YieldRadio\(\)](#) after scheduled TX/RX and instantaneous TX completion, is required, to indicate to the Power Manager that the the radio is no longer busy and can be idled for sleeping.

See [Yielding the Radio](#) for more details.

Definition at line 5925 of file `common/rail.h`

RAIL_GetSchedulerStatus

RAIL_SchedulerStatus_t RAIL_GetSchedulerStatus ([RAIL_Handle_t](#) railHandle)

Get the status of the RAIL scheduler.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- [RAIL_SchedulerStatus_t](#) status.

This function can only be called from a callback context after the [RAIL_EVENT_SCHEDULER_STATUS](#) event occurs.

Definition at line 5936 of file common/rail.h

RAIL_GetSchedulerStatusAlt

```
RAIL_Status_t RAIL_GetSchedulerStatusAlt (RAIL_Handle_t railHandle, RAIL_SchedulerStatus_t * pSchedulerStatus, RAIL_Status_t * pRailStatus)
```

Get the status of the RAIL scheduler, specific to the radio operation, along with [RAIL_Status_t](#) returned by RAIL API invoked by the RAIL scheduler.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	pSchedulerStatus	An application-provided pointer to store RAIL_SchedulerStatus_t status. Can be NULL as long as pRailStatus is not NULL.
[out]	pRailStatus	An application-provided pointer to store RAIL_Status_t of the RAIL API invoked by the RAIL scheduler. Can be NULL as long as pSchedulerStatus is not NULL.

Returns

- Status code indicating success of the function call.

This function can only be called from a callback context after the [RAIL_EVENT_SCHEDULER_STATUS](#) event occurs.

Definition at line 5955 of file common/rail.h

RAIL_SetTaskPriority

```
RAIL_Status_t RAIL_SetTaskPriority (RAIL_Handle_t railHandle, uint8_t priority, RAIL_TaskType_t taskType)
```

Change the priority of a specified task type in multiprotocol.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	priority	Desired new priority for the railHandle's active task
[in]	taskType	Type of task whose priority should be updated

Returns

- Status code indicating success of the function call.

While the application can use this function however it likes, a major use case is being able to increase an infinite receive priority while receiving a packet. In other words, a given [RAIL_Handle_t](#) can maintain a very low priority background receive, but upon getting a [RAIL_EVENT_RX_SYNC1_DETECT_SHIFT](#) or [RAIL_EVENT_RX_SYNC2_DETECT_SHIFT](#) event, the app can call this function to increase the background RX priority to lower the risk another protocol might preempt during packet reception.

Definition at line 5976 of file common/rail.h

RAIL_GetTransitionTime

```
RAIL_Time_t RAIL_GetTransitionTime (void )
```

Get the time needed to switch between protocols.

Parameters

N/A		

Returns

- [RAIL_Time_t](#) Approximate time, in microseconds, needed to switch between protocols.

Note

- The transition time determines how early the scheduler starts processing a protocol switch relative to a scheduled event. This value is approximate and actual transition time may vary slightly due to system overhead.

Definition at line 5992 of file `common/rail.h`

RAIL_SetTransitionTime

```
void RAIL_SetTransitionTime (RAIL\_Time\_t transitionTime)
```

Set the time needed to switch between protocols.

Parameters

[in]	transitionTime	Time, in microseconds, needed to switch between protocols.

Call this API only once, before any protocol is initialized via [RAIL_Init\(\)](#). Changing this value during normal operation can result in improper scheduling behavior.

Note

- The transition time determines how early the scheduler starts processing a protocol switch relative to a scheduled event. This value is approximate and does not account for all system overhead. Setting this value too low may cause scheduling issues or missed events.

Definition at line 6007 of file `common/rail.h`

RAIL_SchedulerInfo_t

A structure to hold information used by the scheduler.

For multiprotocol versions of RAIL, this can be used to control how a receive or transmit operation is run. It's not necessary in single-protocol applications.

Public Attributes

uint8_t	priority	The scheduler priority to use for this operation.
RAIL_Time_t	slipTime	The amount of time in microseconds that this operation can slip by into the future and still be run.
RAIL_Time_t	transactionTime	The transaction time in microseconds for this operation.

Public Attribute Documentation

priority

uint8_t RAIL_SchedulerInfo_t::priority

The scheduler priority to use for this operation.

This priority is used to preempt a long running lower-priority task to ensure higher-priority operations complete in time. A lower numerical value represents a higher logical priority meaning 0 is the highest priority and 255 is the lowest.

Definition at line 511 of file common/rail_types.h

slipTime

RAIL_Time_t RAIL_SchedulerInfo_t::slipTime

The amount of time in microseconds that this operation can slip by into the future and still be run.

This time is relative to the start time which may be the current time for relative transmits. If the scheduler can't start the operation by this time, it will be considered a failure.

Definition at line 518 of file common/rail_types.h

transactionTime

RAIL_Time_t RAIL_SchedulerInfo_t::transactionTime

The transaction time in microseconds for this operation.

Since transaction times may not be known exactly, use a minimum or an expected guess for this time. The scheduler will use the value entered here to look for overlaps between low-priority and high-priority tasks and attempt to find a schedule where all tasks get to run.

Definition at line 526 of file common/railTypes.h

EFR32xG2x

EFR32xG2x

EFR32xG2x-specific multiprotocol support defines.

Macros

```
#define TRANSITION_TIME_US 500
Time it takes to take care of protocol switching.
```

Slxx3xx

Slxx3xx

Slxx3xx-specific multiprotocol support defines.

Macros

```
#define TRANSITION_TIME_US 450
Time it takes to take care of protocol switching.
```

PTI Packet Trace

PTI Packet Trace

Basic APIs to set up and interact with PTI settings.

These enumerations and structures are used with RAIL PTI API.

EFR32 supports SPI and UART PTI and is configurable in terms of baud rates and PTI pin locations.

Modules

[RAIL_PtiConfig_t](#)

Enumerations

```
enum    RAIL_PtiMode_t {
        RAIL_PTI_MODE_DISABLED = 0
        RAIL_PTI_MODE_SPI = 1
        RAIL_PTI_MODE_UART = 2
        RAIL_PTI_MODE_UART_ONEWIRE = 3
    }
    A channel type enumeration.

enum    RAIL_PtiProtocol_t {
        RAIL_PTI_PROTOCOL_CUSTOM = 0
        RAIL_PTI_PROTOCOL_THREAD = 2
        RAIL_PTI_PROTOCOL_BLE = 3
        RAIL_PTI_PROTOCOL_CONNECT = 4
        RAIL_PTI_PROTOCOL_ZIGBEE = 5
        RAIL_PTI_PROTOCOL_ZWAVE = 6
        RAIL_PTI_PROTOCOL_WISUN = 7
        RAIL_PTI_PROTOCOL_802154 = 8
        RAIL_PTI_PROTOCOL_SIDEWALK = 9
        RAIL_PTI_PROTOCOL_BTC = 10
    }
    The protocol that RAIL outputs via the Packet Trace Interface (PTI).
```

Functions

RAIL_Status_t	RAIL_ConfigPti (RAIL_Handle_t railHandle, const RAIL_PtiConfig_t *ptiConfig) Configure PTI pin locations, serial protocols, and baud rates.
RAIL_Status_t	RAIL_GetPtiConfig (RAIL_Handle_t railHandle, RAIL_PtiConfig_t *ptiConfig) Get the currently-active PTI configuration.
RAIL_Status_t	RAIL_EnablePti (RAIL_Handle_t railHandle, bool enable) Enable Packet Trace Interface (PTI) output of packet data.
RAIL_Status_t	RAIL_SetPtiProtocol (RAIL_Handle_t railHandle, RAIL_PtiProtocol_t protocol) Set a protocol that RAIL outputs on PTI.
RAIL_PtiProtocol_t	RAIL_GetPtiProtocol (RAIL_Handle_t railHandle) Get the protocol that RAIL outputs on PTI.

RAIL_PtiMode_t

RAIL_PtiMode_t

A channel type enumeration.

Enumerator	
RAIL_PTI_MODE_DISABLED	Turn PTI off entirely.
RAIL_PTI_MODE_SPI	8-bit SPI mode.
RAIL_PTI_MODE_UART	8-bit UART mode.
RAIL_PTI_MODE_UART_ONEWIRE	9-bit UART mode.

Definition at line 2421 of file common/rail_types.h

RAIL_PtiProtocol_t

RAIL_PtiProtocol_t

The protocol that RAIL outputs via the Packet Trace Interface (PTI).

Enumerator	
RAIL_PTI_PROTOCOL_CUSTOM	PTI output for a custom protocol.
RAIL_PTI_PROTOCOL_THREAD	PTI output for the Thread protocol.
RAIL_PTI_PROTOCOL_BLE	PTI output for the Bluetooth Smart protocol.
RAIL_PTI_PROTOCOL_CONNECT	PTI output for the Connect protocol.
RAIL_PTI_PROTOCOL_ZIGBEE	PTI output for the Zigbee protocol.
RAIL_PTI_PROTOCOL_ZWAVE	PTI output for the Z-Wave protocol.
RAIL_PTI_PROTOCOL_WISUN	PTI output for the Wi-SUN protocol.
RAIL_PTI_PROTOCOL_802154	PTI output for a custom protocol using a built-in 802.15.4 radio config.
RAIL_PTI_PROTOCOL_SIDEWALK	PTI output for Sidewalk protocol.
RAIL_PTI_PROTOCOL_BTC	PTI output for the Bluetooth Classic protocol.

Definition at line 2473 of file common/rail_types.h

Function Documentation

RAIL_ConfigPti

RAIL_Status_t RAIL_ConfigPti (RAIL_Handle_t railHandle, const RAIL_PtiConfig_t * ptiConfig)

Configure PTI pin locations, serial protocols, and baud rates.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.
[in]	ptiConfig	A non-NULL pointer to the PTI configuration structure to use.

Returns

-

Status code indicating success of the function call.

When this function is called prior to the first `RAIL_Init()` call, the PTI configuration is recorded but activation is deferred to `RAIL_Init()`. When called subsequent to `RAIL_Init()`, the radio should be off (idle) and the PTI configuration is recorded and put into effect immediately unless `RAIL_EnablePti()` had been called just prior to disable PTI, in which case activation is deferred to when `RAIL_EnablePti()` is subsequently called to enable PTI.

Only one PTI configuration that can be active on a radio, regardless of the number of protocols (unless the application updates the configuration upon a protocol switch – RAIL does not save the configuration in a protocol RAIL instance).

Note

- On EFR32 platforms GPIO configuration must be unlocked (see GPIO->LOCK register) to configure or use PTI.

Warnings

- As this function relies on GPIO access and RAIL is meant to run in TrustZone non-secure world, it is not supported if GPIO is configured as secure peripheral and it will return `RAIL_STATUS_INVALID_CALL`.

Definition at line 418 of file `common/rail.h`

RAIL_GetPtiConfig

```
RAIL_Status_t RAIL_GetPtiConfig (RAIL_Handle_t railHandle, RAIL_PtiConfig_t * ptiConfig)
```

Get the currently-active PTI configuration.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.
[out]	ptiConfig	A non-NULL pointer to the configuration structure to be filled in with the active PTI configuration.

Returns

- RAIL status indicating success of the function call.

Although most combinations of configurations can be set, it is safest to call this method after configuration to confirm which values were actually set. This function always returns the active PTI configuration on the radio regardless of the active protocol.

Definition at line 434 of file `common/rail.h`

RAIL_EnablePti

```
RAIL_Status_t RAIL_EnablePti (RAIL_Handle_t railHandle, bool enable)
```

Enable Packet Trace Interface (PTI) output of packet data.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.
[in]	enable	PTI is enabled if true; disabled if false.

Returns

- Status code indicating success of the function call.

Similarly to having only one PTI configuration per radio, PTI can only be enabled or disabled for all protocols. It cannot be individually set to enabled and disabled per protocol (unless the application switches it when the protocol switches – RAIL

does not save this state in a protocol RAIL instance).

PTI should be enabled or disabled only when the radio is off (idle).

Note

- The first call to [RAIL_Init\(\)](#) implicitly enables PTI, but it won't take effect unless or until [RAIL_ConfigPti\(\)](#) has been called with a mode other than the default [RAIL_PTI_MODE_DISABLED](#).

Warnings

- On EFR32 platforms GPIO configuration must be unlocked (see GPIO->LOCK register) to configure or use PTI, otherwise a fault or assert might occur. If GPIO configuration locking is desired, PTI must be disabled beforehand either with this function or with [RAIL_ConfigPti\(\)](#) using [RAIL_PTI_MODE_DISABLED](#).
- As this function relies on GPIO access and RAIL is meant to run in TrustZone non-secure world, it is not supported if GPIO is configured as secure peripheral and it will return [RAIL_STATUS_INVALID_CALL](#).

Definition at line 467 of file common/rail.h

RAIL_SetPtiProtocol

```
RAIL_Status_t RAIL_SetPtiProtocol (RAIL\_Handle\_t railHandle, RAIL\_PtiProtocol\_t protocol)
```

Set a protocol that RAIL outputs on PTI.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	protocol	The enumeration representing which protocol the app is using.

Returns

- Status code indicating success of the function call.

The protocol is output via PTI for each packet. Before any protocol is set, the default value is [RAIL_PTI_PROTOCOL_CUSTOM](#). Use one of the enumeration values so that the Network Analyzer can decode the packet.

Note

- This function cannot be called unless the radio is currently in the [RAIL_RF_STATE_IDLE](#) or [RAIL_RF_STATE_INACTIVE](#) states. For this reason, call this function early on before starting radio operations and not changed later.

Definition at line 487 of file common/rail.h

RAIL_GetPtiProtocol

```
RAIL_PtiProtocol_t RAIL_GetPtiProtocol (RAIL\_Handle\_t railHandle)
```

Get the protocol that RAIL outputs on PTI.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- PTI protocol in use.

Definition at line 496 of file common/rail.h

RAIL_PtiConfig_t

A configuration for PTI.

Public Attributes

RAIL_PtiMode_t	mode Packet Trace mode (UART or SPI).
uint32_t	baud Output baudrate for PTI in Hz.
uint8_t	doutLoc
uint8_t	doutPort Data output (DOUT) GPIO port.
uint8_t	doutPin Data output (DOUT) GPIO pin.
uint8_t	dclkLoc
uint8_t	dclkPort Data clock (DCLK) GPIO port.
uint8_t	dclkPin Data clock (DCLK) GPIO pin.
uint8_t	dframeLoc
uint8_t	dframePort Data frame (DFRAME) GPIO port.
uint8_t	dframePin Data frame (DFRAME) GPIO pin.

Public Attribute Documentation

mode

RAIL_PtiMode_t RAIL_PtiConfig_t::mode

Packet Trace mode (UART or SPI).

Definition at line 2446 of file common/rail_types.h

baud

uint32_t RAIL_PtiConfig_t::baud

Output baudrate for PTI in Hz.

Definition at line 2448 of file common/rail_types.h

doutLoc

```
uint8_t RAIL_PtiConfig_t::doutLoc
```

DeprecatedNo longer used (ignored).

Definition at line 2450 of file common/rail_types.h

doutPort

```
uint8_t RAIL_PtiConfig_t::doutPort
```

Data output (DOUT) GPIO port.

Definition at line 2452 of file common/rail_types.h

doutPin

```
uint8_t RAIL_PtiConfig_t::doutPin
```

Data output (DOUT) GPIO pin.

Definition at line 2454 of file common/rail_types.h

dclkLoc

```
uint8_t RAIL_PtiConfig_t::dclkLoc
```

DeprecatedNo longer used (ignored).

Definition at line 2456 of file common/rail_types.h

dclkPort

```
uint8_t RAIL_PtiConfig_t::dclkPort
```

Data clock (DCLK) GPIO port.

Only used in SPI mode.

Definition at line 2458 of file common/rail_types.h

dclkPin

```
uint8_t RAIL_PtiConfig_t::dclkPin
```

Data clock (DCLK) GPIO pin.

Only used in SPI mode.

Definition at line 2460 of file common/rail_types.h

dframeLoc

```
uint8_t RAIL_PtiConfig_t::dframeLoc
```

DeprecatedNo longer used (ignored).

Definition at line 2462 of file common/rail_types.h

dframePort

```
uint8_t RAIL_PtiConfig_t::dframePort
```

Data frame (DFRAME) GPIO port.

Definition at line 2464 of file common/rail_types.h

dframePin

```
uint8_t RAIL_PtiConfig_t::dframePin
```

Data frame (DFRAME) GPIO pin.

Definition at line 2466 of file common/rail_types.h

Protocol-specific

Protocol-specific

Protocol-Specific RAIL APIs.

Modules

[BLE](#)

[IEEE 802.15.4](#)

[Multi-Level Frequency Modulation](#)

[Sidewalk Radio Configurations](#)

[Wireless M-Bus](#)

[Z-Wave](#)

BLE

BLE

Accelerator routines for Bluetooth Low Energy (BLE).

The APIs in this module configure the radio for BLE operation and provide additional helper routines necessary for normal BLE send/receive that aren't available directly in RAIL. RAIL APIs should be used to set up the application. However, [RAIL_ConfigChannels\(\)](#) should not be called to set up the PHY. Instead, `RAIL_BLE_Config*` APIs should be used to set up the 1 Mbps, 2 Mbps, or Coded PHY configurations needed by the application. These APIs will configure the hardware and also configure the set of valid BLE channels.

To implement a standard BLE link layer, you will also need to handle tight turnaround times and send packets at specific instants. This can all be managed through general RAIL functions, such as [RAIL_StartScheduledTx\(\)](#), [RAIL_ScheduleRx\(\)](#), and [RAIL_SetStateTiming\(\)](#). See RAIL APIs for more useful functions.

A simple example to set up the application to be in BLE mode is shown below. Note that this will put the radio on the first advertising channel with the advertising Access Address. In any full-featured BLE application you will need to use the [RAIL_BLE_ConfigChannelRadioParams\(\)](#) function to change the sync word and other parameters as needed based on your connection.

```
// RAIL handle set at initialization time.
static RAIL_Handle_t gRailHandle = NULL;

static void radioEventHandler(RAIL_Handle_t railHandle,
                             RAIL_Events_t events)
{
    // ... handle RAIL events, e.g., receive and transmit completion
}

// Set the radio to receive on the first BLE advertising channel.
void bleAdvertiseEnable(void)
{
    RAIL_Config_t railCfg = {
        .eventsCallback = &radioEventHandler,
    };

    // Initializes the RAIL library and any internal state it requires.
    gRailHandle = RAIL_Init(&railCfg, NULL);

    // Calls the BLE initialization function to load the right radio configuration.
    RAIL_BLE_Init(gRailHandle);

    // Always choose the Viterbi PHY configuration if available on your chip
    // for performance reasons.
    RAIL_BLE_ConfigPhy1MbpsViterbi(gRailHandle);

    // Configures us for the first advertising channel (Physical: 0, Logical: 37).
    // The CRC init value and Access Address come from the BLE specification.
    RAIL_BLE_ConfigChannelRadioParams(gRailHandle,
                                       0x555555,
                                       0x8E89BED6,
                                       37,
                                       false);

    // Starts receiving on physical channel 0 (logical channel 37).
    RAIL_StartRx(gRailHandle, 0, NULL);
}
```

Modules

[RAIL_BLE_State_t](#)

[Angle of Arrival/Departure](#)

[BLE Radio Configurations](#)

[BLE TX Channel Hopping](#)

Enumerations

```
enum    RAIL_BLE_Coding_t {
    RAIL_BLE_Coding_125kbps = 0
    RAIL_BLE_Coding_125kbps_DSA = 1
    RAIL_BLE_Coding_500kbps = 2
    RAIL_BLE_Coding_500kbps_DSA = 3
}
The variant of the BLE Coded PHY.
```

```
enum RAIL_BLE_Phy_t {
    RAIL_BLE_1Mbps = 0U
    RAIL_BLE_2Mbps = 1U
    RAIL_BLE_Coded125kbps = 2U
    RAIL_BLE_Coded500kbps = 3U
    RAIL_BLE_Simulscan = 4U
    RAIL_BLE_AOX2Mbps = 5U
    RAIL_BLE_Quuppa1Mbps = 6U
    RAIL_BLE_CS1Mbps = 7U
    RAIL_BLE_CS2Mbps = 8U
}
The variant of the BLE PHY.
```

```
enum RAIL_BLE_SignalIdentifierMode_t {
    RAIL_BLE_SIGNAL_IDENTIFIER_MODE_DISABLE = 0
    RAIL_BLE_SIGNAL_IDENTIFIER_MODE_1MBPS = 1
    RAIL_BLE_SIGNAL_IDENTIFIER_MODE_2MBPS = 2
}
Available Signal Identifier modes.
```

Functions

RAIL_Status_t	RAIL_BLE_Init (RAIL_Handle_t railHandle) Configure RAIL to run in BLE mode.
RAIL_Status_t	RAIL_BLE_Deinit (RAIL_Handle_t railHandle) Take RAIL out of BLE mode.
bool	RAIL_BLE_IsEnabled (RAIL_Handle_t railHandle) Determine whether BLE mode is enabled or not.
RAIL_Status_t	RAIL_BLE_ConfigPhyQuuppa (RAIL_Handle_t railHandle) Switch to the 1 Mbps Quuppa PHY.
RAIL_Status_t	RAIL_BLE_ConfigPhy1MbpsViterbi (RAIL_Handle_t railHandle) Switch to the Viterbi 1 Mbps BLE PHY.
RAIL_Status_t	RAIL_BLE_ConfigPhy1Mbps (RAIL_Handle_t railHandle) Switch to the legacy non-Viterbi 1 Mbps BLE PHY.
RAIL_Status_t	RAIL_BLE_ConfigPhy2MbpsViterbi (RAIL_Handle_t railHandle) Switch to the Viterbi 2 Mbps BLE PHY.
RAIL_Status_t	RAIL_BLE_ConfigPhy2Mbps (RAIL_Handle_t railHandle) Switch to the legacy non-Viterbi 2 Mbps BLE PHY.
RAIL_Status_t	RAIL_BLE_ConfigPhyCoded (RAIL_Handle_t railHandle, RAIL_BLE_Coding_t bleCoding) Switch to the BLE Coded PHY.
RAIL_Status_t	RAIL_BLE_ConfigPhySimulscan (RAIL_Handle_t railHandle) Switch to the Simulscan PHY.
RAIL_Status_t	RAIL_BLE_ConfigChannelRadioParams (RAIL_Handle_t railHandle, uint32_t crclnit, uint32_t accessAddress, uint16_t channel, bool disableWhitening) Change BLE radio parameters.
RAIL_Status_t	RAIL_BLE_PhySwitchToRx (RAIL_Handle_t railHandle, RAIL_BLE_Phy_t phy, uint16_t railChannel, RAIL_Time_t startRxTime, uint32_t crclnit, uint32_t accessAddress, uint16_t logicalChannel, bool disableWhitening) Change the current BLE PHY and go into receive.

RAIL_Status_t	RAIL_BLE_ConfigSignalIdentifier (RAIL_Handle_t railHandle, RAIL_BLE_SignalIdentifierMode_t signalIdentifierMode) Configure and enable signal identifier for BLE signal detection.
RAIL_Status_t	RAIL_BLE_EnableSignalDetection (RAIL_Handle_t railHandle, bool enable) Enable or disable signal identifier interrupt for BLE signal detection.

Macros

#define	RAIL_BLE_RX_SUBPHY_ID_500K (0U) RAIL_RxPacketDetails_t::subPhyId indicating a 500 kbps packet.
#define	RAIL_BLE_RX_SUBPHY_ID_125K (1U) RAIL_RxPacketDetails_t::subPhyId indicating a 125 kbps packet.
#define	RAIL_BLE_RX_SUBPHY_ID_1M (2U) RAIL_RxPacketDetails_t::subPhyId value indicating a 1 Mbps packet.
#define	RAIL_BLE_RX_SUBPHY_ID_INVALID (3U) RAIL_RxPacketDetails_t::subPhyId invalid value.
#define	RAIL_BLE_RX_SUBPHY_COUNT (4U) The total count of BLE subPhyId's.
#define	RAIL_BLE_EnableSignalIdentifier RAIL_BLE_EnableSignalDetection Backward compatible name for the RAIL_BLE_EnableSignalDetection API.

Enumeration Documentation

RAIL_BLE_Coding_t

RAIL_BLE_Coding_t

The variant of the BLE Coded PHY.

Enumerator	
RAIL_BLE_Coding_125kbps	Enables the 125 kbps variant of the BLE Coded PHY.
RAIL_BLE_Coding_125kbps_DSA	Deprecated Will be removed in a future version of RAIL.
RAIL_BLE_Coding_500kbps	Enables the 500 kbps variant of the BLE Coded PHY.
RAIL_BLE_Coding_500kbps_DSA	Deprecated Will be removed in a future version of RAIL.

Definition at line 116 of file protocol/ble/rail_ble.h

RAIL_BLE_Phy_t

RAIL_BLE_Phy_t

The variant of the BLE PHY.

Enumerator	
RAIL_BLE_1Mbps	BLE 1 Mbps PHY.
RAIL_BLE_2Mbps	BLE 2 Mbps PHY.
RAIL_BLE_Coded125kbps	BLE 125 kbps coded PHY.

RAIL_BLE_Coded500kbps	BLE 500 kbps coded PHY.
RAIL_BLE_Simulscan	BLE Simulscan PHY.
RAIL_BLE_AOX2mbps	BLE 2 Mbps PHY with AoX functionality.
RAIL_BLE_Quuppa1Mbps	BLE 1 Mbps Quuppa PHY.
RAIL_BLE_CS1Mbps	BLE 1 Mbps PHY with CS.
RAIL_BLE_CS2Mbps	BLE 2 Mbps PHY with CS.

Definition at line 139 of file protocol/ble/rail_ble.h

RAIL_BLE_SignalIdentifierMode_t

RAIL_BLE_SignalIdentifierMode_t

Available Signal Identifier modes.

Enumerator	
RAIL_BLE_SIGNAL_IDENTIFIER_MODE_DISABLE	Disable signal detection mode.
RAIL_BLE_SIGNAL_IDENTIFIER_MODE_1MBPS	BLE 1 Mbps (GFSK) detection mode.
RAIL_BLE_SIGNAL_IDENTIFIER_MODE_2MBPS	BLE 2 Mbps (GFSK) detection mode.

Definition at line 275 of file protocol/ble/rail_ble.h

Function Documentation

RAIL_BLE_Init

RAIL_Status_t RAIL_BLE_Init (RAIL_Handle_t railHandle)

Configure RAIL to run in BLE mode.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- Status code indicating success of the function call.

This function changes your radio, channel configuration, and other parameters to match what is needed for BLE, initially establishing the BLE 1 Mbps PHY. To switch back to a default RAIL mode, call [RAIL_BLE_Deinit\(\)](#) first. This function will configure the protocol output on PTI to [RAIL_PT1_PROTOCOL_BLE](#).

Note

- BLE may not be enabled while Auto-ACKing is enabled.

Definition at line 327 of file protocol/ble/rail_ble.h

RAIL_BLE_Deinit

RAIL_Status_t RAIL_BLE_Deinit (RAIL_Handle_t railHandle)

Take RAIL out of BLE mode.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- Status code indicating success of the function call.

This function will undo some of the configuration that happens when you call [RAIL_BLE_Init\(\)](#). After this you can safely run your normal radio initialization code to use a non-BLE configuration. This function does **not** change back your radio or channel configurations so you must do this by manually reinitializing. This also resets the protocol output on PTI to [RAIL_PT_PROTOCOL_CUSTOM](#).

Definition at line 342 of file `protocol/ble/rail_ble.h`

RAIL_BLE_IsEnabled

```
bool RAIL_BLE_IsEnabled (RAIL_Handle_t railHandle)
```

Determine whether BLE mode is enabled or not.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- true if BLE mode is enabled and false otherwise.

This function returns the current status of RAIL's BLE mode. It is enabled by a call to [RAIL_BLE_Init\(\)](#) and disabled by a call to [RAIL_BLE_Deinit\(\)](#).

Definition at line 353 of file `protocol/ble/rail_ble.h`

RAIL_BLE_ConfigPhyQuuppa

```
RAIL_Status_t RAIL_BLE_ConfigPhyQuuppa (RAIL_Handle_t railHandle)
```

Switch to the 1 Mbps Quuppa PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- Status code indicating success of the function call.

You can use this function to switch to the Quuppa PHY.

Note

- Not all chips support the 1 Mbps Quuppa PHY. This API should return [RAIL_STATUS_INVALID_CALL](#) if unsupported by the hardware we're building for.

Definition at line 366 of file `protocol/ble/rail_ble.h`

RAIL_BLE_ConfigPhy1MbpsViterbi

RAIL_Status_t RAIL_BLE_ConfigPhy1MbpsViterbi ([RAIL_Handle_t](#) railHandle)

Switch to the Viterbi 1 Mbps BLE PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- Status code indicating success of the function call.

Use this function to switch back to the default BLE 1 Mbps PHY if you have switched to the 2 Mbps or another configuration. You may only call this function after initializing BLE and while the radio is idle.

Definition at line 378 of file `protocol/ble/rail_ble.h`

RAIL_BLE_ConfigPhy1Mbps

RAIL_Status_t RAIL_BLE_ConfigPhy1Mbps ([RAIL_Handle_t](#) railHandle)

Switch to the legacy non-Viterbi 1 Mbps BLE PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- Status code indicating success of the function call.

Use this function to switch back to the legacy BLE 1 Mbps PHY if you have switched to the 2 Mbps or another configuration. You may only call this function after initializing BLE and while the radio is idle.

DeprecatedBLE non-Viterbi PHYs are no longer supported.

Definition at line 394 of file `protocol/ble/rail_ble.h`

RAIL_BLE_ConfigPhy2MbpsViterbi

RAIL_Status_t RAIL_BLE_ConfigPhy2MbpsViterbi ([RAIL_Handle_t](#) railHandle)

Switch to the Viterbi 2 Mbps BLE PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- Status code indicating success of the function call.

Use this function to switch back to the BLE 2 Mbps PHY from the default 1 Mbps option. You may only call this function after initializing BLE and while the radio is idle.

Definition at line 408 of file `protocol/ble/rail_ble.h`

RAIL_BLE_ConfigPhy2Mbps

RAIL_Status_t RAIL_BLE_ConfigPhy2Mbps ([RAIL_Handle_t](#) railHandle)

Switch to the legacy non-Viterbi 2 Mbps BLE PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- Status code indicating success of the function call.

Use this function to switch back to legacy BLE 2 Mbps PHY from the default 1 Mbps option. You may only call this function after initializing BLE and while the radio is idle.

DeprecatedBLE non-Viterbi PHYs are no longer supported.

Definition at line 424 of file `protocol/ble/rail_ble.h`

RAIL_BLE_ConfigPhyCoded

RAIL_Status_t RAIL_BLE_ConfigPhyCoded ([RAIL_Handle_t](#) railHandle, [RAIL_BLE_Coding_t](#) bleCoding)

Switch to the BLE Coded PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.
[in]	bleCoding	The RAIL_BLE_Coding_t to use

Returns

- Status code indicating success of the function call.

Use this function to switch back to BLE Coded PHY from the default 1 Mbps option. You may only call this function after initializing BLE and while the radio is idle. When using a BLE Coded PHY, the [RAIL_RxPacketDetails_t::subPhyId](#) marks the coding of the received packet. A subPhyId of 0 marks a 500 kbps packet, and a subPhyId of 1 marks a 125 kbps packet.

Definition at line 442 of file `protocol/ble/rail_ble.h`

RAIL_BLE_ConfigPhySimulscan

RAIL_Status_t RAIL_BLE_ConfigPhySimulscan ([RAIL_Handle_t](#) railHandle)

Switch to the Simulscan PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- Status code indicating success of the function call.

Use this function to switch to the BLE Simulscan PHY. You may only call this function after initializing BLE and while the radio is idle. When using Simulscan PHY, the `RAIL_RxPacketDetails_t::subPhyId` marks the coding of the received packet. A subPhyId of 0 marks a 500 kbps packet, a subPhyId of 1 marks a 125 kbps packet, and a subPhyId of 2 marks a 1 Mbps packet.

Note

- : The Simulscan PHY is supported only on some parts. The preprocessor symbol `RAIL_BLE_SUPPORTS_SIMULSCAN_PHY` and the runtime function `RAIL_BLE_SupportsSimulscanPhy()` may be used to test for support of the Simulscan PHY.

Definition at line 463 of file `protocol/ble/rail_ble.h`

RAIL_BLE_ConfigChannelRadioParams

RAIL_Status_t RAIL_BLE_ConfigChannelRadioParams (`RAIL_Handle_t` railHandle, uint32_t crclnit, uint32_t accessAddress, uint16_t channel, bool disableWhitening)

Change BLE radio parameters.

Parameters

[in]	railHandle	A handle for RAIL instance.
[in]	crclnit	The value to use for CRC initialization.
[in]	accessAddress	The access address to use for the connection. The bits of this parameter are transmitted or received LSB first.
[in]	channel	The logical channel that you're changing to, which initializes the whitener if used.
[in]	disableWhitening	This can turn off the whitening engine and is useful for sending BLE test mode packets that don't have this turned on.

Returns

- Status code indicating success of the function call.

This function can be used to switch radio parameters on every connection and/or channel change. It is BLE-aware and will set the access address, preamble, CRC initialization value, and whitening configuration without requiring you to load a new radio configuration. This function should be called after switching to a particular BLE phy (1 Mbps, 2 Mbps, etc.) and not while the radio is active.

Definition at line 517 of file `protocol/ble/rail_ble.h`

RAIL_BLE_PhySwitchToRx

RAIL_Status_t RAIL_BLE_PhySwitchToRx (`RAIL_Handle_t` railHandle, `RAIL_BLE_Phy_t` phy, uint16_t railChannel, `RAIL_Time_t` startRxTime, uint32_t crclnit, uint32_t accessAddress, uint16_t logicalChannel, bool disableWhitening)

Change the current BLE PHY and go into receive.

Parameters

[in]	railHandle	A handle for RAIL instance.
[in]	phy	Indicates which PHY to receive on.
[in]	railChannel	Which channel of the given PHY to receive on.
[in]	startRxTime	Absolute near-future RAIL time to enter RX.
[in]	crclnit	The value to use for CRC initialization.

[in]	accessAddress	The access address to use for the connection. The bits of this parameter are transmitted or received LSB first.
[in]	logicalChannel	The logical channel that you're changing to, which initializes the whitener if used.
[in]	disableWhitening	This can turn off the whitening engine and is useful for sending BLE test mode packets that don't have this turned on.

Returns

- Status code indicating success of the function call.

This function is used to implement auxiliary packet reception, as defined in the BLE specification. The radio will be put into IDLE, the PHY and channel will be changed, and then receive will be entered at the start time given. The new receive will have a timeout of 30 us, which means that this function should only be called if the offset unit is 30 us.

This function is extremely time-sensitive, and may only be called within the interrupt context of a [RAIL_EVENT_RX_PACKET_RECEIVED](#) event.

Definition at line 548 of file `protocol/ble/rail_ble.h`

RAIL_BLE_ConfigSignalIdentifier

```
RAIL_Status_t RAIL_BLE_ConfigSignalIdentifier (RAIL_Handle_t railHandle, RAIL_BLE_SignalIdentifierMode_t signalIdentifierMode)
```

Configure and enable signal identifier for BLE signal detection.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	signalIdentifierMode	Mode of signal identifier operation.

Returns

- Status code indicating success of the function call.

This features allows detection of BLE signal on air based on the mode. This function must be called once before [RAIL_BLE_EnableSignalDetection](#) to configure and enable signal identifier.

To enable event for signal detection [RAIL_ConfigEvents\(\)](#) must be called for enabling [RAIL_EVENT_SIGNAL_DETECTED](#).

This function is only supported by chips where [RAIL_BLE_SUPPORTS_SIGNAL_IDENTIFIER](#) and [RAIL_BLE_SupportsSignalIdentifier\(\)](#) are true.

Definition at line 577 of file `protocol/ble/rail_ble.h`

RAIL_BLE_EnableSignalDetection

```
RAIL_Status_t RAIL_BLE_EnableSignalDetection (RAIL_Handle_t railHandle, bool enable)
```

Enable or disable signal identifier interrupt for BLE signal detection.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	Signal detection is enabled if true, disabled if false.

Returns

Status code indicating success of the function call.

[RAIL_BLE_ConfigSignalIdentifier\(\)](#) must be called once before calling this function to configure and enable signal identifier. Once a signal is detected signal detection will be turned off and this function should be called to re-enable the signal detection without needing to call [RAIL_BLE_ConfigSignalIdentifier\(\)](#) if the signal identifier is already configured and enabled.

This function is only supported by chips where [RAIL_BLE_SUPPORTS_SIGNAL_IDENTIFIER](#) and [RAIL_BLE_SupportsSignalIdentifier\(\)](#) are true.

Definition at line 598 of file protocol/ble/rail_ble.h

RAIL_BLE_State_t

A structure for BLE radio state parameters.

Public Attributes

uint32_t	crclnit	The value used to initialize the CRC algorithm.
uint32_t	accessAddress	The access address used for the connection.
uint16_t	channel	The logical channel used.
bool	disableWhitening	Indicates whether the whitening engine should be off (generally used for testing only).
uint16_t	whiteInit	Reserved for future use; specify 0.

Public Attribute Documentation

crclnit

uint32_t RAIL_BLE_State_t::crclnit

The value used to initialize the CRC algorithm.

Definition at line 297 of file protocol/ble/rail_ble.h

accessAddress

uint32_t RAIL_BLE_State_t::accessAddress

The access address used for the connection.

It is transmitted or received least-significant bit first.

Definition at line 302 of file protocol/ble/rail_ble.h

channel

uint16_t RAIL_BLE_State_t::channel

The logical channel used.

Definition at line 304 of file protocol/ble/rail_ble.h

disableWhitening

bool RAIL_BLE_State_t::disableWhitening

Indicates whether the whitening engine should be off (generally used for testing only).

Definition at line 306 of file protocol/ble/rail_ble.h

whitelnit

uint16_t RAIL_BLE_State_t::whitelnit

Reserved for future use; specify 0.

Definition at line 308 of file protocol/ble/rail_ble.h

Angle of Arrival/Departure

Angle of Arrival/Departure

These APIs are to a stack implementing BLE's angle of arrival and angle of departure functionality.

They are designed for use by the Silicon Labs BLE stack only at this time and may cause problems if accessed directly.

Modules

[RAIL_BLE_AoxConfig_t](#)

[RAIL_BLE_AoxAntennaPortPins_t](#)

[RAIL_BLE_AoxAntennaConfig_t](#)

Enumerations

```
enum RAIL\_BLE\_AoxOptions\_t {
    RAIL_BLE_AOX_OPTIONS_SAMPLE_MODE_SHIFT = 0
    RAIL_BLE_AOX_OPTIONS_CONNLESS_SHIFT = 1
    RAIL_BLE_AOX_OPTIONS_CONN_SHIFT = 2
    RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK_SHIFT = 3
}
```

Angle of Arrival/Departure options bit fields.

Functions

bool	RAIL_BLE_LockCteBuffer (RAIL_Handle_t railHandle, bool lock)	Lock/unlock the CTE buffer from the application's perspective.
bool	RAIL_BLE_CteBufferIsLocked (RAIL_Handle_t railHandle)	Determine whether the CTE buffer is currently locked or not.
uint8_t	RAIL_BLE_GetCteSampleOffset (RAIL_Handle_t railHandle)	Get the offset into CTE sample of CTE data.
uint32_t	RAIL_BLE_GetCteSampleRate (RAIL_Handle_t railHandle)	Get the effective sample rate used by the ADC to capture the CTE samples.
RAIL_Status_t	RAIL_BLE_ConfigAox (RAIL_Handle_t railHandle, const RAIL_BLE_AoxConfig_t *aoxConfig)	Configure Angle of Arrival/Departure (AoX) functionality.
RAIL_Status_t	RAIL_BLE_InitCte (RAIL_Handle_t railHandle)	Perform one time initialization of AoX registers.
RAIL_Status_t	RAIL_BLE_ConfigAoxAntenna (RAIL_Handle_t railHandle, RAIL_BLE_AoxAntennaConfig_t *antennaConfig)	Perform initialization of AoX antenna GPIO pins.

Macros

```
#define RAIL\_BLE\_AOX\_ANTENNA\_PIN\_COUNT (6U)
```

The maximum number of GPIO pins used for AoX Antenna switching.

```
#define RAIL_BLE_AOX_OPTIONS_DO_SWITCH (0U)

#define RAIL_BLE_AOX_OPTIONS_TX_ENABLED (0U)

#define RAIL_BLE_AOX_OPTIONS_RX_ENABLED (0U)

#define RAIL_BLE_AOX_OPTIONS_LOCK_CTE_BUFFER_SHIFT
RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK_SHIFT

#define RAIL_BLE_AOX_OPTIONS_DISABLED (0U)
Disable the AoX feature.

#define RAIL_BLE_AOX_OPTIONS_SAMPLE_MODE (1U << RAIL_BLE_AOX_OPTIONS_SAMPLE_MODE_SHIFT)
Sets one of the two AoX sampling/switching modes: 1 us or 2 us window.

#define RAIL_BLE_AOX_OPTIONS_CONNLESS (1U << RAIL_BLE_AOX_OPTIONS_CONNLESS_SHIFT)
Enables connectionless AoX Rx packets.

#define RAIL_BLE_AOX_OPTIONS_CONN (1U << RAIL_BLE_AOX_OPTIONS_CONN_SHIFT)
Enables connection based AoX Rx packets.

#define RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK (1U <<
RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK_SHIFT)
Disables CTE buffer lock.

#define RAIL_BLE_AOX_OPTIONS_ENABLED (RAIL_BLE_AOX_OPTIONS_CONN |
RAIL_BLE_AOX_OPTIONS_CONNLESS)
Enables connection based or connectionless AoX Rx packets.
```

Enumeration Documentation

RAIL_BLE_AoxOptions_t

RAIL_BLE_AoxOptions_t

Angle of Arrival/Departure options bit fields.

Enumerator	
RAIL_BLE_AOX_OPTIONS_SAMPLE_MODE_SHIFT	Shift position of RAIL_BLE_AOX_OPTIONS_SAMPLE_MODE bit.
RAIL_BLE_AOX_OPTIONS_CONNLESS_SHIFT	Shift position of RAIL_BLE_AOX_OPTIONS_CONNLESS bit.
RAIL_BLE_AOX_OPTIONS_CONN_SHIFT	Shift position of RAIL_BLE_AOX_OPTIONS_CONN bit.
RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK_SHIFT	Shift position of RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK bit.

Definition at line 647 of file protocol/ble/rail_ble.h

Function Documentation

RAIL_BLE_LockCteBuffer

bool RAIL_BLE_LockCteBuffer ([RAIL_Handle_t](#) railHandle, bool lock)

Lock/unlock the CTE buffer from the application's perspective.

Parameters

[in]	railHandle	A RAIL instance handle.

[in]	lock	Lock the CTE buffer if true and unlock it if false.

The radio will write to the buffer only if the bit is NOT set at the beginning of the sampling period. The radio will set the bit once the sampling period starts to indicate that some CTE data has been collected, which will not be overwritten during the next sampling period, unless the buffer is unlocked by the application.

Returns

- true if the CTE buffer is locked after the call, otherwise false.

Definition at line 771 of file protocol/ble/rail_ble.h

RAIL_BLE_CteBufferIsLocked

```
bool RAIL_BLE_CteBufferIsLocked (RAIL_Handle_t railHandle)
```

Determine whether the CTE buffer is currently locked or not.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- true if CTE buffer is locked and false otherwise.

Definition at line 779 of file protocol/ble/rail_ble.h

RAIL_BLE_GetCteSampleOffset

```
uint8_t RAIL_BLE_GetCteSampleOffset (RAIL_Handle_t railHandle)
```

Get the offset into CTE sample of CTE data.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- The offset of CTE data in a CTE sample in bytes. On unsupported platforms this returns 0.

Definition at line 788 of file protocol/ble/rail_ble.h

RAIL_BLE_GetCteSampleRate

```
uint32_t RAIL_BLE_GetCteSampleRate (RAIL_Handle_t railHandle)
```

Get the effective sample rate used by the ADC to capture the CTE samples.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- The actual sample rate used to capture the CTE in samples per second. On unsupported platforms this returns 0.

Definition at line 797 of file protocol/ble/rail_ble.h

RAIL_BLE_ConfigAox

RAIL_Status_t RAIL_BLE_ConfigAox (RAIL_Handle_t railHandle, const RAIL_BLE_AoxConfig_t * aoxConfig)

Configure Angle of Arrival/Departure (AoX) functionality.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	aoxConfig	Configuration options for AoX

Returns

- Status code indicating success of the function call.

AoX is a method of radio localization which infers angle of arrival/departure of the signal based on different phases of the raw I/Q signal from different antennas by controlling external RF switch during the continuous tone extension (CTE). Connection based AoX packets are different than normal BLE packets in that they have 3 header bytes instead of 2 and they have CTE appended after the payload's CRC. 3rd byte or CTE info contains CTE length. Connectionless AoX packets have 2 header bytes and CTE info is part of the payload.

Note

- Calling [RAIL_GetRadioEntropy\(\)](#) during AoX reception may break packet reception.

Definition at line 818 of file protocol/ble/rail_ble.h

RAIL_BLE_InitCte

RAIL_Status_t RAIL_BLE_InitCte (RAIL_Handle_t railHandle)

Perform one time initialization of AoX registers.

Parameters

[in]	railHandle	A RAIL instance handle.

This function must be called before [RAIL_BLE_ConfigAox\(\)](#) and before configuring the BLE PHY.

Returns

- Status code indicating success of the function call.

Definition at line 829 of file protocol/ble/rail_ble.h

RAIL_BLE_ConfigAoxAntenna

RAIL_Status_t RAIL_BLE_ConfigAoxAntenna (RAIL_Handle_t railHandle, RAIL_BLE_AoxAntennaConfig_t * antennaConfig)

Perform initialization of AoX antenna GPIO pins.

Parameters

[in]	railHandle	A RAIL instance handle.

[in]	antennaConfig	A pointer to the antenna configuration structure to hold the set of GPIO ports and pins for AoX antenna switching.

This function must be called before calls to [RAIL_BLE_InitCte\(\)](#) and [RAIL_BLE_ConfigAox\(\)](#), and before configuring the BLE PHY, else a [RAIL_STATUS_INVALID_CALL](#) is returned.

If user configures more pins in [RAIL_BLE_AoxAntennaConfig_t::antCount](#) than allowed by [RAIL_BLE_AOX_ANTENNA_PIN_COUNT](#), then the API returns [RAIL_STATUS_INVALID_PARAMETER](#).

If user configures lesser than or equal to number of pins allowed by [RAIL_BLE_AOX_ANTENNA_PIN_COUNT](#), then the requested number of pins are configured and [RAIL_STATUS_NO_ERROR](#) is returned.

If AoX antenna switching is inactive, non-AoX transmits and receives will occur on the first antenna specified by the antenna pattern or on the default antenna if no antenna pattern is provided.

Note

- This function must only be called while channel sounding is disabled. If channel sounding has been enabled [RAIL_STATUS_INVALID_STATE](#) will be returned.

Returns

- Status code indicating success of the function call.

Note

- Use the compile time symbol [RAIL_BLE_SUPPORTS_ANTENNA_SWITCHING](#) or the runtime call [RAIL_BLE_SupportsAntennaSwitching\(\)](#) to check whether the platform supports this feature.

Warnings

- As this function relies on GPIO access and RAIL is meant to run in TrustZone non-secure world, it is not supported if GPIO is configured as secure peripheral and it will return [RAIL_STATUS_INVALID_CALL](#).

Definition at line 867 of file protocol/ble/rail_ble.h

RAIL_BLE_AoxConfig_t

Contains arguments for `RAIL_BLE_ConfigAox()` function.

Public Attributes

<code>RAIL_BLE_AoxOptions_t</code>	<code>aoxOptions</code> AoX options.
<code>uint16_t</code>	<code>cteBuffSize</code> Size of the raw AoX CTE (continuous tone extension) data capture buffer in bytes.
<code>uint32_t *</code>	<code>cteBuffAddr</code> Address to where the received CTE is written.
<code>uint8_t *</code>	<code>antArrayAddr</code> Address to first element of antenna pattern array.
<code>uint8_t</code>	<code>antArraySize</code> Number of entries in the antenna pattern array.

Public Attribute Documentation

aoxOptions

```
RAIL_BLE_AoxOptions_t RAIL_BLE_AoxConfig_t::aoxOptions
```

AoX options.

Definition at line 706 of file `protocol/ble/rail_ble.h`

cteBuffSize

```
uint16_t RAIL_BLE_AoxConfig_t::cteBuffSize
```

Size of the raw AoX CTE (continuous tone extension) data capture buffer in bytes.

Note this value should be a multiple of 4 as each IQ sample requires 4 bytes.

Definition at line 712 of file `protocol/ble/rail_ble.h`

cteBuffAddr

```
uint32_t* RAIL_BLE_AoxConfig_t::cteBuffAddr
```

Address to where the received CTE is written.

Buffer must be 32-bit aligned.

Definition at line 717 of file `protocol/ble/rail_ble.h`

antArrayAddr

```
uint8_t* RAIL_BLE_AoxConfig_t::antArrayAddr
```

Address to first element of antenna pattern array.

Array must be in RAM. Each element of the array contains an antenna number. The switching pattern is defined by the order of antennas in this array.

Definition at line 723 of file `protocol/ble/rail_ble.h`

antArraySize

```
uint8_t RAIL_BLE_AoxConfig_t::antArraySize
```

Number of entries in the antenna pattern array.

Definition at line 727 of file `protocol/ble/rail_ble.h`

RAIL_BLE_AoxAntennaPortPins_t

Contains elements of [RAIL_BLE_AoxAntennaConfig_t](#) struct.

Public Attributes

- uint8_t

[antPort](#)

The port which is used for AoX antenna switching.
- uint8_t

[antPin](#)

The pin which is used for AoX antenna switching.

Public Attribute Documentation

antPort

```
uint8_t RAIL_BLE_AoxAntennaPortPins_t::antPort
```

The port which is used for AoX antenna switching.

Definition at line 736 of file `protocol/ble/rail_ble.h`

antPin

```
uint8_t RAIL_BLE_AoxAntennaPortPins_t::antPin
```

The pin which is used for AoX antenna switching.

Definition at line 738 of file `protocol/ble/rail_ble.h`

RAIL_BLE_AoxAntennaConfig_t

Contains arguments for [RAIL_BLE_ConfigAoxAntenna\(\)](#) function.

Public Attributes

RAIL_BLE_AoxAntennaPortPins_t *	antPortPin A pointer to an array containing struct of port and pin used for AoX antenna switching.
uint8_t	antCount Number of antenna pins to be configured.

Public Attribute Documentation

antPortPin

```
RAIL_BLE_AoxAntennaPortPins_t* RAIL_BLE_AoxAntennaConfig_t::antPortPin
```

A pointer to an array containing struct of port and pin used for AoX antenna switching.

Definition at line 750 of file `protocol/ble/rail_ble.h`

antCount

```
uint8_t RAIL_BLE_AoxAntennaConfig_t::antCount
```

Number of antenna pins to be configured.

Definition at line 754 of file `protocol/ble/rail_ble.h`

BLE Radio Configurations

BLE Radio Configurations

Radio configurations for the RAIL BLE Accelerator.

These radio configurations are used to configure BLE when a function such as `RAIL_BLE_ConfigPhy1MbpsViterbi()` is called. Each radio configuration listed below is compiled into the RAIL library as a weak symbol that will take into account per-die defaults. If the board configuration in use has different settings than the default, such as a different radio subsystem clock frequency, these radio configurations can be overridden to account for those settings.

Variables

<code>const RAIL_ChannelConfig_t *const</code>	<code>RAIL_BLE_Phy1Mbps</code> Default PHY to use for BLE 1 Mbps non-Viterbi.
<code>const RAIL_ChannelConfig_t *const</code>	<code>RAIL_BLE_Phy2Mbps</code> Default PHY to use for BLE 2 Mbps non-Viterbi.
<code>const RAIL_ChannelConfig_t *const</code>	<code>RAIL_BLE_Phy1MbpsViterbi</code> Default PHY to use for BLE 1 Mbps Viterbi.
<code>const RAIL_ChannelConfig_t *const</code>	<code>RAIL_BLE_Phy2MbpsViterbi</code> Default PHY to use for BLE 2 Mbps Viterbi.
<code>const RAIL_ChannelConfig_t *const</code>	<code>RAIL_BLE_Phy2MbpsAox</code> PHY to use for BLE 2 Mbps with AoX functionality.
<code>const RAIL_ChannelConfig_t *const</code>	<code>RAIL_BLE_Phy125kbps</code> Default PHY to use for BLE Coded 125 kbps.
<code>const RAIL_ChannelConfig_t *const</code>	<code>RAIL_BLE_Phy500kbps</code> Default PHY to use for BLE Coded 500 kbps.
<code>const RAIL_ChannelConfig_t *const</code>	<code>RAIL_BLE_PhySimulscan</code> Default PHY to use for BLE Simulscan.
<code>const RAIL_ChannelConfig_t *const</code>	<code>RAIL_BLE_PhyQuuppa</code> Default 1 Mbps Quuppa PHY.

Variable Documentation

RAIL_BLE_Phy1Mbps

```
const RAIL_ChannelConfig_t* const RAIL_BLE_Phy1Mbps
```

Default PHY to use for BLE 1 Mbps non-Viterbi.

Will be NULL if [RAIL_BLE_SUPPORTS_1MBPS_NON_VITERBI](#) is 0.

Definition at line 189 of file `protocol/ble/rail_ble.h`

RAIL_BLE_Phy2Mbps

```
const RAIL_ChannelConfig_t* const RAIL_BLE_Phy2Mbps
```

Default PHY to use for BLE 2 Mbps non-Viterbi.

Will be NULL if [RAIL_BLE_SUPPORTS_2MBPS_NON_VITERBI](#) is 0.

Definition at line 195 of file `protocol/ble/rail_ble.h`

RAIL_BLE_Phy1MbpsViterbi

```
const RAIL_ChannelConfig_t* const RAIL_BLE_Phy1MbpsViterbi
```

Default PHY to use for BLE 1 Mbps Viterbi.

Will be NULL if [RAIL_BLE_SUPPORTS_1MBPS_VITERBI](#) is 0.

Definition at line 201 of file `protocol/ble/rail_ble.h`

RAIL_BLE_Phy2MbpsViterbi

```
const RAIL_ChannelConfig_t* const RAIL_BLE_Phy2MbpsViterbi
```

Default PHY to use for BLE 2 Mbps Viterbi.

Will be NULL if [RAIL_BLE_SUPPORTS_2MBPS_VITERBI](#) is 0.

Definition at line 207 of file `protocol/ble/rail_ble.h`

RAIL_BLE_Phy2MbpsAox

```
const RAIL_ChannelConfig_t* const RAIL_BLE_Phy2MbpsAox
```

PHY to use for BLE 2 Mbps with AoX functionality.

Will be NULL if either [RAIL_BLE_SUPPORTS_2MBPS_VITERBI](#) or [RAIL_BLE_SUPPORTS_AOX](#) is 0.

Definition at line 229 of file `protocol/ble/rail_ble.h`

RAIL_BLE_Phy125kbps

```
const RAIL_ChannelConfig_t* const RAIL_BLE_Phy125kbps
```


Default PHY to use for BLE Coded 125 kbps.

Will be NULL if [RAIL_BLE_SUPPORTS_CODED_PHY](#) is 0. This PHY can receive on both 125 kbps and 500 kbps BLE Coded, but will only transmit at 125 kbps.

Definition at line 236 of file `protocol/ble/rail_ble.h`

RAIL_BLE_Phy500kbps

```
const RAIL_ChannelConfig_t* const RAIL_BLE_Phy500kbps
```

Default PHY to use for BLE Coded 500 kbps.

Will be NULL if [RAIL_BLE_SUPPORTS_CODED_PHY](#) is 0. This PHY can receive on both 125 kbps and 500 kbps BLE Coded, but will only transmit at 125 kbps.

Definition at line 243 of file `protocol/ble/rail_ble.h`

RAIL_BLE_PhySimulscan

```
const RAIL_ChannelConfig_t* const RAIL_BLE_PhySimulscan
```

Default PHY to use for BLE Simulscan.

Will be NULL if [RAIL_BLE_SUPPORTS_SIMULSCAN_PHY](#) is 0. This PHY can receive on 1 Mbps as well as 125 kbps and 500 kbps BLE Coded, but will only transmit at 1 Mbps.

Definition at line 250 of file `protocol/ble/rail_ble.h`

RAIL_BLE_PhyQuuppa

```
const RAIL_ChannelConfig_t* const RAIL_BLE_PhyQuuppa
```

Default 1 Mbps Quuppa PHY.

Will be NULL if [RAIL_BLE_SUPPORTS_QUUPPA](#) is 0.

Definition at line 256 of file `protocol/ble/rail_ble.h`

BLE TX Channel Hopping

BLE TX Channel Hopping

```
// Configuration to send one additional packet
static RAIL_BLE_TxChannelHoppingConfigEntry_t entry[1];
static uint32_t buffer[BUFFER_SIZE];
static RAIL_BLE_TxRepeatConfig_t repeat = {
    .iterations = 1,
    .repeatOptions = RAIL_TX_REPEAT_OPTION_HOP,
    .delayOrHop.channelHopping = {
        .buffer = buffer,
        .bufferLength = BUFFER_SIZE,
        .numberOfChannels = 1,
        .entries = &entry[0],
    },
};

// Send a normal packet on the current channel, then a packet on a new channel
void bleSendThenAdvertise(uint8_t *firstPacket, uint8_t *secondPacket)
{
    // Load both packets into the FIFO
    RAIL_WriteTxFifo(railHandle, firstPacket, FIRST_PACKET_LEN, true);
    RAIL_WriteTxFifo(railHandle, secondPacket, SECOND_PACKET_LEN, false);

    // Configure a 300 us turnaround between transmits
    entry[0].delayMode = RAIL_CHANNEL_HOPPING_DELAY_MODE_STATIC;
    entry[0].delay = 300; // microseconds

    // Use default advertising parameters
    entry[0].disableWhitening = false;
    entry[0].crclnit = 0x00555555;
    entry[0].accessAddress = 0x8E89BED6;

    // Transmit the repeated packet on the first advertising channel
    entry[0].phy = RAIL_BLE_1Mbps;
    entry[0].railChannel = 0;
    entry[0].logicalChannel = 37;

    // Configure repeated transmit in RAIL, then transmit, sending both packets
    RAIL_BLE_SetNextTxRepeat(railHandle, &repeat);
    RAIL_StartTx(railHandle, currentChannel, RAIL_TX_OPTIONS_DEFAULT, NULL);
}
```

Modules

[RAIL_BLE_TxChannelHoppingConfigEntry_t](#)

[RAIL_BLE_TxChannelHoppingConfig_t](#)

[RAIL_BLE_TxRepeatConfig_t](#)

Functions

[RAIL_Status_t](#) [RAIL_BLE_SetNextTxRepeat](#)(RAIL_Handle_t railHandle, const RAIL_BLE_TxRepeatConfig_t *repeatConfig)
Set up automatic repeated transmits after the next transmit.

Function Documentation

RAIL_BLE_SetNextTxRepeat

```
RAIL_Status_t RAIL_BLE_SetNextTxRepeat (RAIL_Handle_t railHandle, const RAIL_BLE_TxRepeatConfig_t * repeatConfig)
```

Set up automatic repeated transmits after the next transmit.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	repeatConfig	A non-NULL pointer to the configuration structure for repeated transmits.

Returns

- Status code indicating a success of the function call.

Repeated transmits will occur after an application-initiated transmit caused by calling one of the [Packet Transmit](#) APIs. The repetition will only occur after the first application-initiated transmit after this function is called. Future repeated transmits must be requested by calling this function again.

Each repeated transmit that occurs will have full [PTI Packet Trace](#) information and will receive events such as [RAIL_EVENT_TX_PACKET_SENT](#) as normal.

If a TX error occurs during the repetition, the process will abort and the TX error transition from [RAIL_SetTxTransitions\(\)](#) will be used. If the repetition completes successfully, the TX success transition from [RAIL_SetTxTransitions\(\)](#) will be used.

Any call to [RAIL_Idle\(\)](#), [RAIL_StopTx\(\)](#), or [RAIL_SetTxTransitions\(\)](#) will clear the pending repeated transmits. The state will also be cleared by another call to this function. To clear the repeated transmits before they've started without stopping other radio actions, call this function with a [RAIL_BLE_TxRepeatConfig_t::iterations](#) count of 0. A DMP switch will clear this state only if the initial transmit triggering the repeated transmits has started.

The application is responsible for populating the transmit data to be used by the repeated transmits via [RAIL_SetTxFifo\(\)](#) or [RAIL_WriteTxFifo\(\)](#). Data will be transmitted from the TX FIFO. If the TX FIFO does not have sufficient data to transmit, a TX error and a [RAIL_EVENT_TX_UNDERFLOW](#) will occur. To avoid an underflow, the application should queue data to be transmitted as early as possible.

This function will fail to configure the repetition if a transmit of any kind is ongoing, including during the time between an initial transmit and the end of a previously-configured repetition.

Note

- Use the compile time symbol [RAIL_SUPPORTS_TX_TO_TX](#) or the runtime call [RAIL_SupportsTxToTx\(\)](#) to check whether the platform supports this feature.

Definition at line 1831 of file `protocol/ble/rail_ble.h`

RAIL_BLE_TxChannelHoppingConfigEntry_t

Structure that represents one of the channels that is part of a [RAIL_BLE_TxChannelHoppingConfig_t](#) sequence of channels used in channel hopping.

Public Attributes

uint32_t	delay	Idle time in microseconds to wait before hopping into the channel indicated by this entry.
RAIL_BLE_Phy_t	phy	The BLE PHY to use for this hop's transmit.
uint8_t	logicalChannel	The logical channel to use for this hop's transmit.
uint8_t	railChannel	The channel number to be used for this hop's transmit.
bool	disableWhitening	This can turn off the whitening engine and is useful for sending BLE test mode packets that don't have this turned on.
uint32_t	crcInit	The value to use for CRC initialization.
uint32_t	accessAddress	The access address to use for the connection.

Public Attribute Documentation

delay

uint32_t RAIL_BLE_TxChannelHoppingConfigEntry_t::delay

Idle time in microseconds to wait before hopping into the channel indicated by this entry.

Definition at line 1682 of file `protocol/ble/rail_ble.h`

phy

RAIL_BLE_Phy_t RAIL_BLE_TxChannelHoppingConfigEntry_t::phy

The BLE PHY to use for this hop's transmit.

Definition at line 1686 of file `protocol/ble/rail_ble.h`

logicalChannel

uint8_t RAIL_BLE_TxChannelHoppingConfigEntry_t::logicalChannel

The logical channel to use for this hop's transmit.

The whitener will be reinitialized if used.

Definition at line 1691 of file protocol/ble/rail_ble.h

railChannel

```
uint8_t RAIL_BLE_TxChannelHoppingConfigEntry_t::railChannel
```

The channel number to be used for this hop's transmit.

If this is an invalid channel for the chosen PHY, the call to [RAIL_SetNextTxRepeat\(\)](#) will fail.

Definition at line 1697 of file protocol/ble/rail_ble.h

disableWhitening

```
bool RAIL_BLE_TxChannelHoppingConfigEntry_t::disableWhitening
```

This can turn off the whitening engine and is useful for sending BLE test mode packets that don't have this turned on.

Definition at line 1702 of file protocol/ble/rail_ble.h

crclnit

```
uint32_t RAIL_BLE_TxChannelHoppingConfigEntry_t::crclnit
```

The value to use for CRC initialization.

Definition at line 1706 of file protocol/ble/rail_ble.h

accessAddress

```
uint32_t RAIL_BLE_TxChannelHoppingConfigEntry_t::accessAddress
```

The access address to use for the connection.

Definition at line 1710 of file protocol/ble/rail_ble.h

RAIL_BLE_TxChannelHoppingConfig_t

Wrapper struct that will contain the sequence of [RAIL_BLE_TxChannelHoppingConfigEntry_t](#) that represents the channel sequence to use during TX Channel Hopping.

Public Attributes

uint32_t *	buffer	Pointer to contiguous global read-write memory that will be used by RAIL to store channel hopping information throughout its operation.
uint16_t	bufferLength	This parameter must be set to the length of the buffer array.
uint8_t	numberOfChannels	The number of channels that is in the channel hopping sequence.
uint8_t	reserved	Pad bytes reserved for future use and currently ignored.
RAIL_BLE_TxChannelHoppingConfigEntry_t *	entries	A pointer to the first element of an array of RAIL_BLE_TxChannelHoppingConfigEntry_t that represents the channels used during channel hopping.

Public Attribute Documentation

buffer

```
uint32_t* RAIL_BLE_TxChannelHoppingConfig_t::buffer
```

Pointer to contiguous global read-write memory that will be used by RAIL to store channel hopping information throughout its operation.

It need not be initialized and applications should never write data anywhere in this buffer.

Definition at line 1726 of file `protocol/ble/rail_ble.h`

bufferLength

```
uint16_t RAIL_BLE_TxChannelHoppingConfig_t::bufferLength
```

This parameter must be set to the length of the buffer array.

This way, during configuration, the software can confirm it's writing within the range of the buffer. The configuration API will return an error if `bufferLength` is insufficient.

Definition at line 1733 of file `protocol/ble/rail_ble.h`

numberOfChannels

```
uint8_t RAIL_BLE_TxChannelHoppingConfig_t::numberOfChannels
```

The number of channels that is in the channel hopping sequence.

Definition at line 1735 of file `protocol/ble/rail_ble.h`

reserved

```
uint8_t RAIL_BLE_TxChannelHoppingConfig_t::reserved
```

Pad bytes reserved for future use and currently ignored.

Definition at line 1739 of file `protocol/ble/rail_ble.h`

entries

```
RAIL_BLE_TxChannelHoppingConfigEntry_t* RAIL_BLE_TxChannelHoppingConfig_t::entries
```

A pointer to the first element of an array of [RAIL_BLE_TxChannelHoppingConfigEntry_t](#) that represents the channels used during channel hopping.

The number of entries in this array must be `numberOfChannels`.

Definition at line 1746 of file `protocol/ble/rail_ble.h`

RAIL_BLE_TxRepeatConfig_t

A configuration structure for repeated transmits.

Public Attributes

<code>uint16_t</code>	iterations	The number of repeated transmits to run.
RAIL_TxRepeatOptions_t	repeatOptions	Repeat option(s) to apply.
RAIL_TransitionTime_t	delay	When RAIL_TX_REPEAT_OPTION_HOP is not set, this specifies the delay time between each repeated transmit.
RAIL_BLE_TxChannelHoppingConfig_t	channelHopping	When RAIL_TX_REPEAT_OPTION_HOP is set, this specifies the channel hopping configuration to use when hopping between repeated transmits.
<code>union</code> <code>RAIL_BLE_TxRepeatConfig_t::@2</code>	delayOrHop	Per-repeat delay or hopping configuration, depending on repeatOptions.

Public Attribute Documentation

iterations

```
uint16_t RAIL_BLE_TxRepeatConfig_t::iterations
```

The number of repeated transmits to run.

A total of (iterations + 1) transmits will go on-air in the absence of errors.

Definition at line 1757 of file protocol/ble/rail_ble.h

repeatOptions

```
RAIL_TxRepeatOptions_t RAIL_BLE_TxRepeatConfig_t::repeatOptions
```

Repeat option(s) to apply.

Definition at line 1761 of file protocol/ble/rail_ble.h

delay

```
RAIL_TransitionTime_t RAIL_BLE_TxRepeatConfig_t::delay
```

When [RAIL_TX_REPEAT_OPTION_HOP](#) is not set, this specifies the delay time between each repeated transmit.

Specify [RAIL_TRANSITION_TIME_KEEP](#) to use the current [RAIL_StateTiming_t::txToTx](#) transition time setting.

Definition at line 1772 of file protocol/ble/rail_ble.h

channelHopping

```
RAIL_BLE_TxChannelHoppingConfig_t RAIL_BLE_TxRepeatConfig_t::channelHopping
```

When [RAIL_TX_REPEAT_OPTION_HOP](#) is set, this specifies the channel hopping configuration to use when hopping between repeated transmits.

Per-hop delays are configured within each [RAIL_BLE_TxChannelHoppingConfigEntry_t::delay](#) rather than this union's delay field.

Definition at line 1780 of file `protocol/ble/rail_ble.h`

delayOrHop

```
union RAIL_BLE_TxRepeatConfig_t::@2 RAIL_BLE_TxRepeatConfig_t::delayOrHop
```

Per-repeat delay or hopping configuration, depending on repeatOptions.

Definition at line 1781 of file `protocol/ble/rail_ble.h`

IEEE 802.15.4

IEEE 802.15.4

IEEE 802.15.4 configuration routines.

The functions in this group configure RAIL IEEE 802.15.4 hardware acceleration which includes IEEE 802.15.4 format filtering, address filtering, Acking, and filtering based on the frame type.

To configure IEEE 802.15.4 functionality, the application must first set up a RAIL instance with [RAIL_Init\(\)](#) and other setup functions. Instead of [RAIL_ConfigChannels\(\)](#), however, an application may use [RAIL_IEEE802154_Config2p4GHzRadio\(\)](#) to set up the official IEEE 2.4 GHz 802.15.4 PHY. This configuration is shown below.

802.15.4 defines its `macAckWaitDuration` from the end of the transmitted packet to complete reception of the Ack. RAIL's `ackTimeout` only covers sync word detection of the Ack. Therefore, subtract the Ack's PHY header and payload time to get RAIL's `ackTimeout` setting. For 2.4 GHz OQPSK, `macAckWaitDuration` is specified as 54 symbols; subtracting 2-symbol PHY header and 10-symbol payload yields a RAIL `ackTimeout` of 42 symbols or 672 microseconds at 16 microseconds/symbol.

```
static RAIL_Handle_t railHandle = NULL; // Initialized somewhere else.

static const RAIL_IEEE802154_Config_t rail154Config = {
    .addresses = NULL,
    .ackConfig = {
        .enable = true, // Turn on auto Ack for IEEE 802.15.4.
        .ackTimeout = 672, // See note above: 54-12 sym * 16 us/sym = 672 us.
        .rxTransitions = {
            .success = RAIL_RF_STATE_RX, // Return to RX after Ack processing
            .error = RAIL_RF_STATE_RX, // Ignored
        },
        .txTransitions = {
            .success = RAIL_RF_STATE_RX, // Return to RX after Ack processing
            .error = RAIL_RF_STATE_RX, // Ignored
        },
    },
    .timings = {
        .idleToRx = 100,
        .idleToTx = 100,
        .rxToTx = 192, // 12 symbols * 16 us/symbol = 192 us
        .txToRx = 192, // 12 symbols * 16 us/symbol = 192 us
        .rxSearchTimeout = 0, // Not used
        .txToRxSearchTimeout = 0, // Not used
    },
    .framesMask = RAIL_IEEE802154_ACCEPT_STANDARD_FRAMES,
    .promiscuousMode = false, // Enable format and address filtering.
    .isPanCoordinator = false,
    .defaultFramePendingInOutgoingAcks = false,
};

void config154(void)
{
    // Configure the radio and channels for 2.4 GHz IEEE 802.15.4.
    RAIL_IEEE802154_Config2p4GHzRadio(railHandle);
    // Initialize the IEEE 802.15.4 configuration using the static configuration above.
    RAIL_IEEE802154_Init(railHandle, &rail154Config);
}
```

To configure address filtering, call [RAIL_IEEE802154_SetAddresses\(\)](#) with a structure containing all addresses or call the individual [RAIL_IEEE802154_SetPanId\(\)](#), [RAIL_IEEE802154_SetShortAddress\(\)](#), and [RAIL_IEEE802154_SetLongAddress\(\)](#) APIs. RAIL supports [RAIL_IEEE802154_MAX_ADDRESSES](#) number of address pairs to receive packets from multiple IEEE 802.15.4 networks at the same time. Broadcast addresses are supported by default without any additional configuration so

they do not consume one of these slots. If the application does not require all address pairs, be sure to set unused ones to the proper disabled value for each type. These can be found in the [RAIL_IEEE802154_AddrConfig_t](#) documentation. Below is an example of setting filtering for one set of addresses.

```
// PAN Id over-the-air value of 0x34 0x12.
// Short Address over-the-air byte order of 0x78 0x56.
// Long address with over-the-air byte order of 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88.

// Set up all addresses simultaneously.
RAIL_Status_t setup1(void)
{
    RAIL_IEEE802154_AddrConfig_t nodeAddress = {
        { 0x1234, 0xFFFF, 0xFFFF },
        { 0x5678, 0xFFFF, 0xFFFF },
        { { 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88 },
          { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
          { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 } }
    };
    return RAIL_IEEE802154_SetAddresses(railHandle, &nodeAddress);
}

// Alternatively, the addresses can be set up individually as follows:
void setup2(void)
{
    RAIL_Status_t status;
    const uint8_t longAddress[] = { 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88 };

    status = RAIL_IEEE802154_SetPanId(railHandle, 0x1234, 0);
    assert(status == RAIL_STATUS_NO_ERROR);
    status = RAIL_IEEE802154_SetShortAddress(railHandle, 0x5678, 0);
    assert(status == RAIL_STATUS_NO_ERROR);
    status = RAIL_IEEE802154_SetLongAddress(railHandle, longAddress, 0);
    assert(status == RAIL_STATUS_NO_ERROR);
}
```

Address filtering will be enabled except when in promiscuous mode, which can be set with [RAIL_IEEE802154_SetPromiscuousMode\(\)](#). The addresses may be changed at runtime. However, if you are receiving a packet while reconfiguring the address filters, you may get undesired behavior so it's safest to do this while not in receive.

Auto Ack is controlled by the [RAIL_IEEE802154_Config_t::ackConfig](#) and [RAIL_IEEE802154_Config_t::timings](#) fields passed to [RAIL_IEEE802154_Init\(\)](#). After initialization, they may be controlled using the normal [Auto-Ack](#) and [State Transitions](#) APIs. When in IEEE 802.15.4 mode, the Ack will generally have a 5 byte length, its Frame Type will be Ack, its Frame Version 0 (2003), and its Frame Pending bit will be false unless the [RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#) event is triggered in which case it will default to the [RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks](#) setting. If the default Frame Pending setting is incorrect, the app must call [RAIL_IEEE802154_ToggleFramePending\(\)](#) (formerly [RAIL_IEEE802154_SetFramePending\(\)](#)) while handling the [RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#) event.

This event must be turned on by the user and will fire whenever a data request is being received so that the stack can determine if there is pending data. Note that if the default Frame Pending bit needs to be changed, it must be done quickly. Otherwise, the Ack may already have been transmitted with the default setting. Check the return code of [RAIL_IEEE802154_ToggleFramePending\(\)](#) to be sure that the bit was changed in time.

Transmit and receive operations are done using the standard RAIL APIs in IEEE 802.15.4 mode. To send packets using the correct CSMA configuration, use [RAIL_CSMA_CONFIG_802_15_4_2003_2p4_GHz_OQPSK_CSMA](#) define that can initialize the `csmaConfig` structure passed to [RAIL_StartCcaCsmatx\(\)](#).

Modules

[RAIL_IEEE802154_Address_t](#)

[RAIL_IEEE802154_AddrConfig_t](#)

[RAIL_IEEE802154_Config_t](#)
[RAIL_IEEE802154_RxChannelSwitchingCfg_t](#)
[RAIL_IEEE802154_ModeSwitchPhr_t](#)
[IEEE 802.15.4 Radio Configurations](#)

Enumerations

```
enum RAIL\_IEEE802154\_AddressLength\_t {
    RAIL_IEEE802154_ShortAddress = 2
    RAIL_IEEE802154_LongAddress = 3
}
Different lengths that an 802.15.4 address can have.
```

```
enum RAIL\_IEEE802154\_PtiRadioConfig\_t {
    RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ = 0x00U
    RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_ANTDIV = 0x01U
    RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_COEX = 0x02U
    RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_ANTDIV_COEX = 0x03U
    RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM = 0x08U
    RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM_ANTDIV = 0x09U
    RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM_COEX = 0x0AU
    RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM_ANTDIV_COEX = 0x0BU
    RAIL_IEEE802154_PTI_RADIO_CONFIG_863MHZ_GB868 = 0x85U
    RAIL_IEEE802154_PTI_RADIO_CONFIG_915MHZ_GB868 = 0x86U
    RAIL_IEEE802154_PTI_RADIO_CONFIG_915MHZ_R23_NA_EXT = 0x97U
}
802.15.4 PTI radio configuration mode
```

```
enum RAIL\_IEEE802154\_EOptions\_t {
    RAIL_IEEE802154_E_OPTION_GB868_SHIFT = 0
    RAIL_IEEE802154_E_OPTION_ENH_ACK_SHIFT
    RAIL_IEEE802154_E_OPTION_IMPLICIT_BROADCAST_SHIFT
}
802.15.4E-2012 options, in reality a bitmask.
```

```
enum RAIL\_IEEE802154\_GOptions\_t {
    RAIL_IEEE802154_G_OPTION_GB868_SHIFT = 0
    RAIL_IEEE802154_G_OPTION_DYNFEC_SHIFT
    RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH_SHIFT
}
802.15.4G-2012 options, in reality a bitmask.
```

```
enum RAIL\_IEEE802154\_CcaMode\_t {
    RAIL_IEEE802154_CCA_MODE_RSSI = 0
    RAIL_IEEE802154_CCA_MODE_SIGNAL
    RAIL_IEEE802154_CCA_MODE_SIGNAL_OR_RSSI
    RAIL_IEEE802154_CCA_MODE_SIGNAL_AND_RSSI
    RAIL_IEEE802154_CCA_MODE_ALWAYS_TRANSMIT
    RAIL_IEEE802154_CCA_MODE_COUNT
}
Available CCA modes.
```

```
enum RAIL\_IEEE802154\_SignalIdentifierMode\_t {
    RAIL_IEEE802154_SIGNAL_IDENTIFIER_MODE_DISABLE = 0
    RAIL_IEEE802154_SIGNAL_IDENTIFIER_MODE_154
}
Available Signal identifier modes.
```

Functions

RAIL_Status_t	RAIL_IEEE802154_Init (RAIL_Handle_t railHandle, const RAIL_IEEE802154_Config_t *config) Initialize RAIL for IEEE802.15.4 features.
RAIL_Status_t	RAIL_IEEE802154_Config2p4GHzRadio (RAIL_Handle_t railHandle) Configure the radio for 2.4 GHz 802.15.4 operation.
RAIL_Status_t	RAIL_IEEE802154_Config2p4GHzRadioAntDiv (RAIL_Handle_t railHandle) Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity.
RAIL_Status_t	RAIL_IEEE802154_Config2p4GHzRadioAntDivCoex (RAIL_Handle_t railHandle) Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity optimized for radio coexistence.
RAIL_Status_t	RAIL_IEEE802154_Config2p4GHzRadioCoex (RAIL_Handle_t railHandle) Configure the radio for 2.4 GHz 802.15.4 operation optimized for radio coexistence.
RAIL_Status_t	RAIL_IEEE802154_Config2p4GHzRadioFem (RAIL_Handle_t railHandle) Configure the radio for 2.4 GHz 802.15.4 operation with a front end module.
RAIL_Status_t	RAIL_IEEE802154_Config2p4GHzRadioAntDivFem (RAIL_Handle_t railHandle) Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity optimized for a front end module.
RAIL_Status_t	RAIL_IEEE802154_Config2p4GHzRadioCoexFem (RAIL_Handle_t railHandle) Configure the radio for 2.4 GHz 802.15.4 operation optimized for radio coexistence and a front end module.
RAIL_Status_t	RAIL_IEEE802154_Config2p4GHzRadioAntDivCoexFem (RAIL_Handle_t railHandle) Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity optimized for radio coexistence and a front end module.
RAIL_Status_t	RAIL_IEEE802154_Config2p4GHzRadioCustom1 (RAIL_Handle_t railHandle) Configure the radio for 2.4 GHz 802.15.4 operation with custom settings.
RAIL_Status_t	RAIL_IEEE802154_ConfigGB863MHzRadio (RAIL_Handle_t railHandle) Configure the radio for Sub-GHz GB868 863 MHz 802.15.4 operation.
RAIL_Status_t	RAIL_IEEE802154_ConfigGB915MHzRadio (RAIL_Handle_t railHandle) Configure the radio for Sub-GHz GB868 915 MHz 802.15.4 operation.
RAIL_Status_t	RAIL_IEEE802154_Deinit (RAIL_Handle_t railHandle) De-initialize IEEE802.15.4 hardware acceleration.
bool	RAIL_IEEE802154_IsEnabled (RAIL_Handle_t railHandle) Return whether IEEE802.15.4 hardware acceleration is currently enabled.
RAIL_IEEE802154_PtiRadioConfig_t	RAIL_IEEE802154_GetPtiRadioConfig (RAIL_Handle_t railHandle) Return IEEE802.15.4 PTI radio config.
RAIL_Status_t	RAIL_IEEE802154_SetAddresses (RAIL_Handle_t railHandle, const RAIL_IEEE802154_AddrConfig_t *addresses) Configure the RAIL Address Filter for 802.15.4 filtering.
RAIL_Status_t	RAIL_IEEE802154_SetPanId (RAIL_Handle_t railHandle, uint16_t panId, uint8_t index) Set a PAN Id for 802.15.4 address filtering.
RAIL_Status_t	RAIL_IEEE802154_SetShortAddress (RAIL_Handle_t railHandle, uint16_t shortAddr, uint8_t index) Set a short address for 802.15.4 address filtering.
RAIL_Status_t	RAIL_IEEE802154_SetLongAddress (RAIL_Handle_t railHandle, const uint8_t *longAddr, uint8_t index) Set a long address for 802.15.4 address filtering.

RAIL_Status_t	RAIL_IEEE802154_SetPanCoordinator (RAIL_Handle_t railHandle, bool isPanCoordinator) Set whether the current node is a PAN coordinator.
RAIL_Status_t	RAIL_IEEE802154_SetPromiscuousMode (RAIL_Handle_t railHandle, bool enable) Set whether to enable 802.15.4 promiscuous mode.
RAIL_Status_t	RAIL_IEEE802154_ConfigEOptions (RAIL_Handle_t railHandle, RAIL_IEEE802154_EOptions_t mask, RAIL_IEEE802154_EOptions_t options) Configure certain 802.15.4E-2012 / 802.15.4-2015 Frame Version 2 features.
RAIL_Status_t	RAIL_IEEE802154_ConfigGOptions (RAIL_Handle_t railHandle, RAIL_IEEE802154_GOptions_t mask, RAIL_IEEE802154_GOptions_t options) Configure certain 802.15.4G-2012 / 802.15.4-2015 SUN PHY features (only for radio configurations designed accordingly).
RAIL_Status_t	RAIL_IEEE802154_ComputeChannelFromPhyModelId (RAIL_Handle_t railHandle, uint8_t newPhyModelId, uint16_t *pChannel) Compute channel to switch to given a targeted PHY Mode Id in the context of Wi-SUN mode switching.
RAIL_Status_t	RAILCb_IEEE802154_IsModeSwitchNewChannelValid (uint32_t currentBaseFreq, uint8_t newPhyModelId, const RAIL_ChannelConfigEntry_t *configEntryNewPhyModelId, uint16_t *pChannel) Manage forbidden channels during mode switch.
RAIL_Status_t	RAIL_IEEE802154_AcceptFrames (RAIL_Handle_t railHandle, uint8_t framesMask) Set which 802.15.4 frame types to accept.
RAIL_Status_t	RAIL_IEEE802154_EnableEarlyFramePending (RAIL_Handle_t railHandle, bool enable) Enable early Frame Pending lookup event notification (RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND).
RAIL_Status_t	RAIL_IEEE802154_EnableDataFramePending (RAIL_Handle_t railHandle, bool enable) Enable Frame Pending lookup event notification (RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND) for MAC Data frames.
RAIL_Status_t	RAIL_IEEE802154_SetFramePending (RAIL_Handle_t railHandle) Change the Frame Pending bit on the outgoing legacy Immediate Ack from the default specified by RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks .
RAIL_Status_t	RAIL_IEEE802154_GetAddress (RAIL_Handle_t railHandle, RAIL_IEEE802154_Address_t *pAddress) Get the source address of the incoming data request.
RAIL_Status_t	RAIL_IEEE802154_WriteEnhAck (RAIL_Handle_t railHandle, const uint8_t *ackData, uint16_t ackDataLen) Write the Auto-Ack FIFO for the next outgoing 802.15.4E Enhanced Ack.
RAIL_Status_t	RAIL_IEEE802154_SetRxToEnhAckTx (RAIL_Handle_t railHandle, RAIL_TransitionTime_t *pRxToEnhAckTx) Set a separate RX packet to TX state transition turnaround time for sending an Enhanced Ack.
uint8_t	RAIL_IEEE802154_ConvertRssiToLqi (uint8_t origLqi, int8_t rssiDbm) Convert RSSI into 802.15.4 Link Quality Indication (LQI) metric compatible with the Silicon Labs Zigbee stack.
uint8_t	RAIL_IEEE802154_ConvertRssiToEd (int8_t rssiDbm) Convert RSSI into 802.15.4 Energy Detection (ED) metric compatible with the Silicon Labs Zigbee stack.
RAIL_Status_t	RAIL_IEEE802154_ConfigSignalIdentifier (RAIL_Handle_t railHandle, RAIL_IEEE802154_SignalIdentifierMode_t signalIdentifierMode) Configure signal identifier for 802.15.4 signal detection.
RAIL_Status_t	RAIL_IEEE802154_EnableSignalDetection (RAIL_Handle_t railHandle, bool enable) Enable or disable signal identifier for 802.15.4 signal detection.
RAIL_Status_t	RAIL_IEEE802154_ConfigCcaMode (RAIL_Handle_t railHandle, RAIL_IEEE802154_CcaMode_t ccaMode) Set 802.15.4 CCA mode.

[RAIL_Status_t](#) [RAIL_IEEE802154_ConfigRxChannelSwitching](#)(RAIL_Handle_t railHandle, const RAIL_IEEE802154_RxChannelSwitchingCfg_t *pConfig)
Configure RX channel switching for 802.15.4.

Macros

#define	RAIL_IEEE802154_MAX_ADDRESSES (3U) The maximum number of allowed addresses of each type.
#define	RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_BYTES (0U) RX channel switching buffer size, in bytes.
#define	RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_ALIGNMENT_TYPE uint32_t Fixed-width type indicating the needed alignment for RX channel switching buffer.
#define	RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_ALIGNMENT (sizeof(RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_ALIGNMENT_TYPE)) Alignment that is needed for RX channel switching buffer.
#define	RAIL_IEEE802154_RX_CHANNEL_SWITCHING_NUM_CHANNELS (2U) Maximum numbers of channels supported for RX channel switching.
#define	RAIL_IEEE802154_E_OPTIONS_NONE 0UL A value representing no options enabled.
#define	RAIL_IEEE802154_E_OPTIONS_DEFAULT RAIL_IEEE802154_E_OPTIONS_NONE All options disabled by default .
#define	RAIL_IEEE802154_E_OPTION_GB868 (1UL << RAIL_IEEE802154_E_OPTION_GB868_SHIFT) An option to enable/disable 802.15.4E-2012 features needed for GB868.
#define	RAIL_IEEE802154_E_OPTION_ENH_ACK (1UL << RAIL_IEEE802154_E_OPTION_ENH_ACK_SHIFT) An option to enable/disable 802.15.4E-2012 features needed for Enhanced Acks.
#define	RAIL_IEEE802154_E_OPTION_IMPLICIT_BROADCAST (1UL << RAIL_IEEE802154_E_OPTION_IMPLICIT_BROADCAST_SHIFT) An option to enable/disable 802.15.4E-2012 macImplicitBroadcast feature.
#define	RAIL_IEEE802154_E_OPTIONS_ALL 0xFFFFFFFFUL A value representing all possible options.
#define	RAIL_IEEE802154_G_OPTIONS_NONE 0UL A value representing no options enabled.
#define	RAIL_IEEE802154_G_OPTIONS_DEFAULT RAIL_IEEE802154_G_OPTIONS_NONE All options disabled by default .
#define	RAIL_IEEE802154_G_OPTION_GB868 (1UL << RAIL_IEEE802154_G_OPTION_GB868_SHIFT) An option to enable/disable 802.15.4G-2012 features needed for GB868.
#define	RAIL_IEEE802154_G_OPTION_DYNFEC (1UL << RAIL_IEEE802154_G_OPTION_DYNFEC_SHIFT) An option to enable/disable 802.15.4G dynamic FEC feature (SUN FSK only).
#define	RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH (1UL << RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH_SHIFT) An option to enable/disable Wi-SUN Mode Switch feature.
#define	RAIL_IEEE802154_G_OPTIONS_ALL 0xFFFFFFFFUL A value representing all possible options.
#define	RAIL_IEEE802154_ACCEPT_BEACON_FRAMES (0x01) When receiving packets, accept 802.15.4 BEACON frame types.
#define	RAIL_IEEE802154_ACCEPT_DATA_FRAMES (0x02)

When receiving packets, accept 802.15.4 DATA frame types.

```
#define RAIL_IEEE802154_ACCEPT_ACK_FRAMES (0x04)
    When receiving packets, accept 802.15.4 Ack frame types.

#define RAIL_IEEE802154_ACCEPT_COMMAND_FRAMES (0x08)
    When receiving packets, accept 802.15.4 COMMAND frame types.

#define RAIL_IEEE802154_ACCEPT_MULTIPURPOSE_FRAMES (0x20)
    When receiving packets, accept 802.15.4-2015 Multipurpose frame types.

#define RAIL_IEEE802154_ACCEPT_STANDARD_FRAMES undefined
    In standard operation, accept BEACON, DATA and COMMAND frames.

#define RAIL_IEEE802154_ToggleFramePending RAIL_IEEE802154_SetFramePending
    Alternate naming for function RAIL\_IEEE802154\_SetFramePending to depict it is used for changing the default setting
    specified by RAIL\_IEEE802154\_Config\_t::defaultFramePendingInOutgoingAcks in an outgoing Ack.

#define RAIL_IEEE802154_EnableSignalIdentifier RAIL_IEEE802154_EnableSignalDetection
    Backward compatible name for the RAIL\_IEEE802154\_EnableSignalDetection API.
```

Enumeration Documentation

RAIL_IEEE802154_AddressLength_t

RAIL_IEEE802154_AddressLength_t

Different lengths that an 802.15.4 address can have.

Enumerator	
RAIL_IEEE802154_ShortAddress	2 byte short address.
RAIL_IEEE802154_LongAddress	8 byte extended address.

Definition at line 185 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_PtiRadioConfig_t

RAIL_IEEE802154_PtiRadioConfig_t

802.15.4 PTI radio configuration mode

Enumerator	
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ	Built-in 2.4 GHz 802.15.4 radio configuration.
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_ANTDIV	Built-in 2.4 GHz 802.15.4 radio configuration with RX antenna diversity support.
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_COEX	Built-in 2.4 GHz 802.15.4 radio configuration optimized for radio coexistence.
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_ANTDIV_COEX	Built-in 2.4 GHz 802.15.4 radio configuration with RX antenna diversity support optimized for radio coexistence.
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM	Built-in 2.4 GHz 802.15.4 radio configuration optimized for front end modules.

RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM_ANTDIV	Built-in 2.4 GHz 802.15.4 radio configuration with RX antenna diversity support optimized for front end modules.
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM_COEX	Built-in 2.4 GHz 802.15.4 radio configuration optimized for radio coexistence and front end modules.
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM_ANTDIV_COEX	Built-in 2.4 GHz 802.15.4 radio configuration with RX antenna diversity support optimized for radio coexistence and front end modules.
RAIL_IEEE802154_PTI_RADIO_CONFIG_863MHZ_GB868	Built-in 863 MHz GB868 802.15.4 radio configuration.
RAIL_IEEE802154_PTI_RADIO_CONFIG_915MHZ_GB868	Built-in 915 MHz GB868 802.15.4 radio configuration.
RAIL_IEEE802154_PTI_RADIO_CONFIG_915MHZ_R23_NA_EXT	External 915 MHz Zigbee R23 802.15.4 NA radio configuration.

Definition at line 802 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_EOptions_t

RAIL_IEEE802154_EOptions_t

802.15.4E-2012 options, in reality a bitmask.

Enumerator

RAIL_IEEE802154_E_OPTION_GB868_SHIFT	Shift position of RAIL_IEEE802154_E_OPTION_GB868 bit.
RAIL_IEEE802154_E_OPTION_ENH_ACK_SHIFT	Shift position of RAIL_IEEE802154_E_OPTION_ENH_ACK bit.
RAIL_IEEE802154_E_OPTION_IMPLICIT_BROADCAST_SHIFT	Shift position of RAIL_IEEE802154_E_OPTION_IMPLICIT_BROADCAST bit.

Definition at line 1001 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_GOptions_t

RAIL_IEEE802154_GOptions_t

802.15.4G-2012 options, in reality a bitmask.

Enumerator

RAIL_IEEE802154_G_OPTION_GB868_SHIFT	Shift position of RAIL_IEEE802154_G_OPTION_GB868 bit.
RAIL_IEEE802154_G_OPTION_DYNFEC_SHIFT	Shift position of RAIL_IEEE802154_G_OPTION_DYNFEC bit.
RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH_SHIFT	Shift position of RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH bit.

Definition at line 1122 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_CcaMode_t

RAIL_IEEE802154_CcaMode_t

Available CCA modes.

Enumerator

RAIL_IEEE802154_CCA_MODE_RSSI	RSSI-based CCA.
RAIL_IEEE802154_CCA_MODE_SIGNAL	Signal Identifier-based CCA.
RAIL_IEEE802154_CCA_MODE_SIGNAL_OR_RSSI	RSSI or signal identifier-based CCA.
RAIL_IEEE802154_CCA_MODE_SIGNAL_AND_RSSI	RSSI and signal identifier-based CCA.
RAIL_IEEE802154_CCA_MODE_ALWAYS_TRANSMIT	ALOHA.
RAIL_IEEE802154_CCA_MODE_COUNT	Number of CCA modes.

Definition at line 1558 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_SignalIdentifierMode_t

RAIL_IEEE802154_SignalIdentifierMode_t

Available Signal identifier modes.

Enumerator

RAIL_IEEE802154_SIGNAL_IDENTIFIER_MODE_DISABLE	Disable signal detection mode.
RAIL_IEEE802154_SIGNAL_IDENTIFIER_MODE_154	2.4 GHz 802.15.4 signal detection mode.

Definition at line 1608 of file protocol/ieee802154/rail_ieee802154.h

Function Documentation

RAIL_IEEE802154_Init

RAIL_Status_t RAIL_IEEE802154_Init ([RAIL_Handle_t](#) railHandle, const [RAIL_IEEE802154_Config_t](#) * config)

Initialize RAIL for IEEE802.15.4 features.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	A non-NULL pointer to an IEEE802154 configuration structure.

Returns

- Status code indicating success of the function call.

This function calls the following RAIL functions to configure the radio for IEEE802.15.4 features.

Initializes the following:

- Enables IEEE802154 hardware acceleration
- Configures RAIL Auto Ack functionality
- Configures RAIL Address Filter for 802.15.4 address filtering

It saves having to call the following functions individually:

- [RAIL_ConfigAutoAck\(\)](#)
- [RAIL_SetRxTransitions\(\)](#)
- [RAIL_SetTxTransitions\(\)](#)
- [RAIL_WriteAutoAckFifo\(\)](#)
- [RAIL_SetStateTiming\(\)](#)
- [RAIL_ConfigAddressFilter\(\)](#)

[RAIL_EnableAddressFilter\(\)](#)

It must be called before most of the RAIL_IEEE802154_* functions.

Definition at line 502 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_Config2p4GHzRadio

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadio ([RAIL_Handle_t](#) railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation. It takes the place of calling [RAIL_ConfigChannels\(\)](#). After this call, channels 11-26 will be available, giving the frequencies of those channels on channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all [RAIL_IEEE802154_GOptions_t](#).

Definition at line 518 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_Config2p4GHzRadioAntDiv

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioAntDiv ([RAIL_Handle_t](#) railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation, but with a configuration that supports antenna diversity. It takes the place of calling [RAIL_ConfigChannels\(\)](#). After this call, channels 11-26 will be available, giving the frequencies of those channels on channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all [RAIL_IEEE802154_GOptions_t](#).

Definition at line 534 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_Config2p4GHzRadioAntDivCoex

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioAntDivCoex ([RAIL_Handle_t](#) railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity optimized for radio coexistence.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation, but with a configuration that supports antenna diversity optimized for radio coexistence. It takes the place of calling [RAIL_ConfigChannels\(\)](#). After this call, channels 11-26 will be available, giving the frequencies of those channels on channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all [RAIL_IEEE802154_GOptions_t](#).

Definition at line 551 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_Config2p4GHzRadioCoex

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioCoex ([RAIL_Handle_t](#) railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation optimized for radio coexistence.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation, but with a configuration that supports radio coexistence. It takes the place of calling [RAIL_ConfigChannels\(\)](#). After this call, channels 11-26 will be available, giving the frequencies of those channels on channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all [RAIL_IEEE802154_GOptions_t](#).

Definition at line 567 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_Config2p4GHzRadioFem

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioFem ([RAIL_Handle_t](#) railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation with a front end module.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation, but with a configuration that supports a front end module. It takes the place of calling [RAIL_ConfigChannels\(\)](#). After this call, channels 11-26 will be available, giving the frequencies of those channels on

channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all [RAIL_IEEE802154_GOptions_t](#).

Definition at line 583 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Config2p4GHzRadioAntDivFem

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioAntDivFem ([RAIL_Handle_t](#) railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity optimized for a front end module.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation, but with a configuration that supports antenna diversity and a front end module. It takes the place of calling [RAIL_ConfigChannels\(\)](#). After this call, channels 11-26 will be available, giving the frequencies of those channels on channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all [RAIL_IEEE802154_GOptions_t](#).

Definition at line 600 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Config2p4GHzRadioCoexFem

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioCoexFem ([RAIL_Handle_t](#) railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation optimized for radio coexistence and a front end module.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation, but with a configuration that supports radio coexistence and a front end module. It takes the place of calling [RAIL_ConfigChannels\(\)](#). After this call, channels 11-26 will be available, giving the frequencies of those channels on channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all [RAIL_IEEE802154_GOptions_t](#).

Definition at line 617 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Config2p4GHzRadioAntDivCoexFem

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioAntDivCoexFem ([RAIL_Handle_t](#) railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity optimized for radio coexistence and a front end module.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation, but with a configuration that supports antenna diversity, radio coexistence and a front end module. It takes the place of calling [RAIL_ConfigChannels\(\)](#). After this call, channels 11-26 will be available, giving the frequencies of those channels on channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all [RAIL_IEEE802154_GOptions_t](#).

Definition at line 634 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Config2p4GHzRadioCustom1

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioCustom1 ([RAIL_Handle_t](#) railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation with custom settings.

Parameters

[in]	railHandle	A RAIL instance handle.

It enables better interoperability with some proprietary PHYs, but doesn't guarantee data sheet performance.

Returns

- Status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation with custom settings. It replaces needing to call [RAIL_ConfigChannels\(\)](#). Do not call this function unless instructed by Silicon Labs.

Note

- This feature is only available on platforms where [RAIL_IEEE802154_SUPPORTS_CUSTOM1_PHY](#) is true.

Definition at line 735 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ConfigGB863MHzRadio

RAIL_Status_t RAIL_IEEE802154_ConfigGB863MHzRadio ([RAIL_Handle_t](#) railHandle)

Configure the radio for Sub-GHz GB868 863 MHz 802.15.4 operation.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

This initializes the radio for Sub-GHz GB868 863 MHz operation. It takes the place of calling [RAIL_ConfigChannels\(\)](#). After this call, GB868 channels in the 863 MHz band (channel pages 28, 29, and 30 – logical channels 0x80..0x9A, 0xA0..0xA8, 0xC0..0xDA, respectively) will be available, as defined by Rev 22 of the Zigbee Specification, 2017 document 05-3474-22, section D.10.2.1.3.2.

Note

- This call implicitly enables [RAIL_IEEE802154_G_OPTION_GB868](#).

Definition at line 754 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_ConfigGB915MHzRadio

RAIL_Status_t RAIL_IEEE802154_ConfigGB915MHzRadio ([RAIL_Handle_t](#) railHandle)

Configure the radio for Sub-GHz GB868 915 MHz 802.15.4 operation.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

This initializes the radio for Sub-GHz GB868 915 MHz operation. It takes the place of calling [RAIL_ConfigChannels\(\)](#). After this call, GB868 channels in the 915 MHz band (channel page 31 – logical channels 0xE0..0xFA) will be available, as defined by Rev 22 of the Zigbee Specification, 2017 document 05-3474-22, section D.10.2.1.3.2.

Note

- This call implicitly enables [RAIL_IEEE802154_G_OPTION_GB868](#).

Definition at line 770 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_Deinit

RAIL_Status_t RAIL_IEEE802154_Deinit ([RAIL_Handle_t](#) railHandle)

De-initialize IEEE802.15.4 hardware acceleration.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

Disables and resets all IEEE802.15.4 hardware acceleration features. This function should only be called when the radio is IDLE. This calls the following:

- [RAIL_SetStateTiming\(\)](#), to reset all timings to 100 us
- [RAIL_EnableAddressFilter\(\)](#) passing false for its enable parameter
- [RAIL_ResetAddressFilter\(\)](#)

Definition at line 785 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_IsEnabled

```
bool RAIL_IEEE802154_IsEnabled (RAIL_Handle_t railHandle)
```

Return whether IEEE802.15.4 hardware acceleration is currently enabled.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if IEEE802.15.4 hardware acceleration was enabled to start with and false otherwise.

Definition at line 794 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_GetPtiRadioConfig

```
RAIL_IEEE802154_PtiRadioConfig_t RAIL_IEEE802154_GetPtiRadioConfig (RAIL_Handle_t railHandle)
```

Return IEEE802.15.4 PTI radio config.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- PTI (Packet Trace Information) radio config Id.

Definition at line 877 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_SetAddresses

```
RAIL_Status_t RAIL_IEEE802154_SetAddresses (RAIL_Handle_t railHandle, const RAIL_IEEE802154_AddrConfig_t * addresses)
```

Configure the RAIL Address Filter for 802.15.4 filtering.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	addresses	A pointer to the address information that should be used.

Returns

- Status code indicating success of the function call. If this returns an error, the 802.15.4 address filter is in an undefined state.

Set up the 802.15.4 address filter to accept messages to the given addresses. This will return false if any of the addresses failed to be set. If NULL is passed in for addresses, all addresses will be set to their reset value.

Definition at line 904 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_SetPanId

RAIL_Status_t RAIL_IEEE802154_SetPanId (RAIL_Handle_t railHandle, uint16_t panId, uint8_t index)

Set a PAN Id for 802.15.4 address filtering.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	panId	The 16-bit PAN Id information. This will be matched against the destination PAN Id of incoming messages. The PAN Id is sent little endian over the air, meaning panId[7:0] is first in the payload followed by panId[15:8]. Set to 0xFFFF to disable for this index.
[in]	index	Indicates which PAN Id to set. Must be below RAIL_IEEE802154_MAX_ADDRESSES.

Returns

- Status code indicating success of the function call.

Set up the 802.15.4 address filter to accept messages to the given PAN Id.

Definition at line 921 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_SetShortAddress

RAIL_Status_t RAIL_IEEE802154_SetShortAddress (RAIL_Handle_t railHandle, uint16_t shortAddr, uint8_t index)

Set a short address for 802.15.4 address filtering.

Parameters

[in]	railHandle	A handle of RAIL instance
[in]	shortAddr	16 bit short address value. This will be matched against the destination short address of incoming messages. The short address is sent little endian over the air meaning shortAddr[7:0] is first in the payload followed by shortAddr[15:8]. Set to 0xFFFF to disable for this index.
[in]	index	Which short address to set. Must be below RAIL_IEEE802154_MAX_ADDRESSES.

Returns

- Status code indicating success of the function call.

Set up the 802.15.4 address filter to accept messages to the given short address.

Definition at line 940 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_SetLongAddress

RAIL_Status_t RAIL_IEEE802154_SetLongAddress (RAIL_Handle_t railHandle, const uint8_t * longAddr, uint8_t index)

Set a long address for 802.15.4 address filtering.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	longAddr	A pointer to an 8-byte array containing the long address information. The long address must be in over-the-air byte order. This will be matched against the destination long address of incoming messages. Set to 0x00 00 00 00 00 00 00 00 to disable for this index.

[in]	index	Indicates which long address to set. Must be below RAIL_IEEE802154_MAX_ADDRESSES .

Returns

- Status code indicating success of the function call.
- Set up the 802.15.4 address filter to accept messages to the given long address.

Definition at line 959 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_SetPanCoordinator

RAIL_Status_t RAIL_IEEE802154_SetPanCoordinator ([RAIL_Handle_t](#) railHandle, bool isPanCoordinator)

Set whether the current node is a PAN coordinator.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	isPanCoordinator	true if this device is a PAN coordinator.

Returns

- Status code indicating success of the function call.
- If the device is a PAN Coordinator, it will accept data and command frames with no destination address. This function will fail if 802.15.4 hardware acceleration is not currently enabled by calling [RAIL_IEEE802154_Init\(\)](#). This setting may be changed at any time when 802.15.4 hardware acceleration is enabled.

Definition at line 976 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_SetPromiscuousMode

RAIL_Status_t RAIL_IEEE802154_SetPromiscuousMode ([RAIL_Handle_t](#) railHandle, bool enable)

Set whether to enable 802.15.4 promiscuous mode.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	true if all frames and addresses should be accepted.

Returns

- Status code indicating success of the function call.
- If promiscuous mode is enabled, no frame or address filtering steps will be performed other than checking the CRC. This function will fail if 802.15.4 hardware acceleration is not currently enabled by calling [RAIL_IEEE802154_Init\(\)](#). This setting may be changed at any time when 802.15.4 hardware acceleration is enabled.

Definition at line 992 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ConfigEOptions

RAIL_Status_t RAIL_IEEE802154_ConfigEOptions ([RAIL_Handle_t](#) railHandle, [RAIL_IEEE802154_EOptions_t](#) mask, [RAIL_IEEE802154_EOptions_t](#) options)

Configure certain 802.15.4E-2012 / 802.15.4-2015 Frame Version 2 features.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mask	A bitmask containing which options should be modified.
[in]	options	A bitmask containing desired options settings. Bit positions for each option are found in the RAIL_IEEE802154_EOptions_t .

Returns

- Status code indicating success of the function call.

This function will fail if 802.15.4 hardware acceleration is not currently enabled by calling [RAIL_IEEE802154_Init\(\)](#) or the platform does not support the feature(s). These settings may be changed at any time when 802.15.4 hardware acceleration is enabled.

Definition at line 1112 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_ConfigGOptions

```
RAIL_Status_t RAIL_IEEE802154_ConfigGOptions (RAIL_Handle_t railHandle, RAIL_IEEE802154_GOptions_t mask, RAIL_IEEE802154_GOptions_t options)
```

Configure certain 802.15.4G-2012 / 802.15.4-2015 SUN PHY features (only for radio configurations designed accordingly).

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mask	A bitmask containing which options should be modified.
[in]	options	A bitmask containing desired options settings. Bit positions for each option are found in the RAIL_IEEE802154_GOptions_t .

Returns

- Status code indicating success of the function call.

This function will fail if 802.15.4 hardware acceleration is not currently enabled by calling [RAIL_IEEE802154_Init\(\)](#), the platform does not support the feature(s), the radio configuration is not appropriate, or the radio is not idle.

Definition at line 1224 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_ComputeChannelFromPhyModeId

```
RAIL_Status_t RAIL_IEEE802154_ComputeChannelFromPhyModeId (RAIL_Handle_t railHandle, uint8_t newPhyModeId, uint16_t * pChannel)
```

Compute channel to switch to given a targeted PHY Mode Id in the context of Wi-SUN mode switching.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	newPhyModeId	A targeted PHY Mode Id.
[out]	pChannel	A pointer to the channel to switch to.

Returns

Status code indicating success of the function call.

This function will fail if:

- the targeted PHY Mode Id is the same as the current PHY Mode Id
- called on a platform that lacks [RAIL_IEEE802154_SUPPORTS_G_MODESWITCH](#)
- called on a platform that doesn't have 802154G options enabled by [RAIL_IEEE802154_ConfigGOptions\(\)](#). For newPhyModelId associated with a FSK FEC_off PHY, if dynamic FEC is activated (see [RAIL_IEEE802154_G_OPTION_DYNFEC](#)), the returned channel can correspond to the associated FSK FEC_on PHY corresponding then to PHY Mode Id = newPhyModelId + 16

Definition at line 1271 of file protocol/ieee802154/rail_ieee802154.h

RAILCb_IEEE802154_IsModeSwitchNewChannelValid

```
RAIL_Status_t RAILCb_IEEE802154_IsModeSwitchNewChannelValid (uint32_t currentBaseFreq, uint8_t newPhyModelId,
const RAIL\_ChannelConfigEntry\_t * configEntryNewPhyModelId, uint16_t * pChannel)
```

Manage forbidden channels during mode switch.

Parameters

[in]	currentBaseFreq	The current frequency of the base channel.
[in]	newPhyModelId	A targeted PHY Mode Id.
[in]	configEntryNewPhyModelId	A pointer to RAIL_ChannelConfigEntry_t structure corresponding to the new PHY configEntry.
[inout]	pChannel	A pointer to the channel to switch to. If channel is valid, the function must just return. If channel is forbidden, the function must update it with the closest valid channel. The highest channel must be selected in case of two valid channels being equidistant to a forbidden channel.

Returns

- Status code indicating success of the function call. It must return [RAIL_STATUS_INVALID_PARAMETER](#) for failure or [RAIL_STATUS_NO_ERROR](#) for success.

This function must fail if no valid channel has been found. If so, RAIL will abort the mode switch.

Note

- This callback will only be called on platforms where [RAIL_IEEE802154_SUPPORTS_G_MODESWITCH](#) is true, [RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH](#) was successfully enabled, and a valid mode switch PHY header is received.

Definition at line 1301 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_AcceptFrames

```
RAIL_Status_t RAIL_IEEE802154_AcceptFrames (RAIL\_Handle\_t railHandle, uint8_t framesMask)
```

Set which 802.15.4 frame types to accept.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	framesMask	A mask containing which 802.15.4 frame types to receive.

Returns

Status code indicating success of the function call.

This function will fail if 802.15.4 hardware acceleration is not currently enabled by calling [RAIL_IEEE802154_Init\(\)](#) or framesMask requests an unsupported frame type. This setting may be changed at any time when 802.15.4 hardware acceleration is enabled. Only Beacon, Data, Ack, Command, and Multipurpose frames may be received. The RAIL_IEEE802154_ACCEPT_*_FRAMES defines may be combined to create a bitmask to pass into this function.

[RAIL_IEEE802154_ACCEPT_ACK_FRAMES](#) behaves slightly different than the other defines. If [RAIL_IEEE802154_ACCEPT_ACK_FRAMES](#) is set, the radio will accept an Ack frame during normal packet reception, but only a truly expected Ack will have its [RAIL_RxPacketDetails_t::isAck](#) true. If [RAIL_IEEE802154_ACCEPT_ACK_FRAMES](#) is not set, Ack frames will be filtered unless they're expected when the radio is waiting for an Ack.

Definition at line 1353 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_EnableEarlyFramePending

RAIL_Status_t RAIL_IEEE802154_EnableEarlyFramePending ([RAIL_Handle_t](#) railHandle, bool enable)

Enable early Frame Pending lookup event notification ([RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#)).

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	true to enable, false to disable.

Returns

- Status code indicating success of the function call.

Normally, [RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#) is triggered after receiving the entire MAC header and MAC command byte for an Ack-requesting MAC command frame. Version 0/1 frames also require that command to be a Data Request for this event to occur. Enabling this feature causes this event to be triggered earlier to allow for more time to determine the type of Ack needed (Immediate or Enhanced) and/or perform frame pending lookup to influence the outgoing Ack by using [RAIL_IEEE802154_WriteEnhAck\(\)](#) or [RAIL_IEEE802154_ToggleFramePending\(\)](#).

For Frame Version 0/1 packets and for Frame Version 2 packets when [RAIL_IEEE802154_E_OPTION_ENH_ACK](#) is not in use, "early" means right after receiving the source address information in the MAC header.

For Frame Version 2 packets when [RAIL_IEEE802154_E_OPTION_ENH_ACK](#) is in use, "early" means right after receiving any Auxiliary Security header which follows the source address information in the MAC header.

This feature is useful when the protocol knows an Ack-requesting MAC Command must be a data poll without needing to receive the MAC Command byte, giving it a bit more time to adjust Frame Pending or generate an Enhanced Ack.

This function will fail if 802.15.4 hardware acceleration is not currently enabled by calling [RAIL_IEEE802154_Init\(\)](#), or on platforms that do not support this feature. This setting may be changed at any time when 802.15.4 hardware acceleration is enabled.

Definition at line 1392 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_EnableDataFramePending

RAIL_Status_t RAIL_IEEE802154_EnableDataFramePending ([RAIL_Handle_t](#) railHandle, bool enable)

Enable Frame Pending lookup event notification ([RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#)) for MAC Data frames.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	true to enable, false to disable.

Returns

- Status code indicating success of the function call.

Normally [RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#) is triggered only for Ack-requesting MAC command frames. Enabling this feature causes this event to also be triggered for MAC data frames, at the same point in the packet as [RAIL_IEEE802154_EnableEarlyFramePending\(\)](#) would trigger. This feature is necessary to support the Thread Basil-Hayden Enhanced Frame Pending feature in Version 0/1 frames, and to support Version 2 Data frames which require an Enhanced Ack.

This function will fail if 802.15.4 hardware acceleration is not currently enabled by calling [RAIL_IEEE802154_Init\(\)](#). This setting may be changed at any time when 802.15.4 hardware acceleration is enabled.

Definition at line 1417 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_SetFramePending

RAIL_Status_t RAIL_IEEE802154_SetFramePending ([RAIL_Handle_t](#) railHandle)

Change the Frame Pending bit on the outgoing legacy Immediate Ack from the default specified by [RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks](#).

Parameters

[in]	railHandle	A handle of RAIL instance

Returns

- Status code indicating success of the function call.

This function must only be called while processing the [RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#) if the Ack for this packet should go out with its Frame Pending bit set differently than what was specified by [RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks](#).

It's intended only for use with 802.15.4 legacy immediate Acks and not 802.15.4E enhanced Acks. This will return [RAIL_STATUS_INVALID_STATE](#) if it is too late to modify the outgoing Immediate Ack.

Definition at line 1439 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_GetAddress

RAIL_Status_t RAIL_IEEE802154_GetAddress ([RAIL_Handle_t](#) railHandle, [RAIL_IEEE802154_Address_t](#) * pAddress)

Get the source address of the incoming data request.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	pAddress	A pointer to RAIL_IEEE802154_Address_t structure to populate with source address information.

Returns

- Status code indicating success of the function call.

This function must only be called when handling the [RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#) event. This will return [RAIL_STATUS_INVALID_STATE](#) if the address information is stale (i.e., it is too late to affect the outgoing Ack).

Definition at line 1462 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_WriteEnhAck

```
RAIL_Status_t RAIL_IEEE802154_WriteEnhAck (RAIL_Handle_t railHandle, const uint8_t * ackData, uint16_t ackDataLen)
```

Write the Auto-Ack FIFO for the next outgoing 802.15.4E Enhanced Ack.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	ackData	A pointer to Ack data to transmit This may be NULL, in which case it's assumed the data has already been emplaced into the Ack buffer and RAIL just needs to be told how many bytes are there. Use RAIL_GetAutoAckFifo() to get the address of RAIL's Auto-Ack buffer in RAM and its size.
[in]	ackDataLen	Length of Ack data, in bytes. If this exceeds RAIL_AUTOACK_MAX_LENGTH the function will return RAIL_STATUS_INVALID_PARAMETER .

Returns

- Status code indicating success of the function call.

This function sets the Auto-Ack data to use in acknowledging the frame being received. It must only be called while processing the [RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#), and is intended for use when packet information from [RAIL_GetRxIncomingPacketInfo\(\)](#) indicates an 802.15.4E Enhanced Ack must be sent instead of a legacy Immediate Ack. [RAIL_IEEE802154_ToggleFramePending\(\)](#) should not be called for an Enhanced Ack; instead the Enhanced Ack's Frame Control Field should have the Frame Pending bit set appropriately in its ackData. This will return [RAIL_STATUS_INVALID_STATE](#) if it is too late to write the outgoing Ack – a situation that will likely trigger a [RAIL_EVENT_TXACK_UNDERFLOW](#) event. When successful, the Enhanced ackData will only be sent once. Subsequent packets needing an Enhanced Ack will each need to call this function to write their Ack information.

Definition at line 1493 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_SetRxToEnhAckTx

```
RAIL_Status_t RAIL_IEEE802154_SetRxToEnhAckTx (RAIL_Handle_t railHandle, RAIL_TransitionTime_t * pRxToEnhAckTx)
```

Set a separate RX packet to TX state transition turnaround time for sending an Enhanced Ack.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pRxToEnhAckTx	A pointer to the turnaround transition requested for Enhanced Acks. It will be updated with the actual time set. Requesting a time of 0 will sync the Enhanced Ack turnaround time with that used for immediate Acks (and output 0). Requesting a time of RAIL_TRANSITION_TIME_KEEP will output the current Enhanced Ack timing parameter (0 if it is the same as that used for Immediate Acks).

Returns

- Status code indicating a success of the function call. An error will not update the pRxToEnhAckTx output parameter.

Normally Immediate and Enhanced Acks are both sent using the [RAIL_IEEE802154_Config_t::timings](#) rxToTx turnaround time. If the stack needs more time to prepare an Enhanced Ack, it can call this function after [RAIL_IEEE802154_Init\(\)](#) to set a longer turnaround time used just for Enhanced Ack transmits.

This function will fail on platforms that lack [RAIL_IEEE802154_SUPPORTS_E_ENHANCED_ACK](#).

Definition at line 1520 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ConvertRssiToLqi

uint8_t RAIL_IEEE802154_ConvertRssiToLqi (uint8_t origLqi, int8_t rssiDbm)

Convert RSSI into 802.15.4 Link Quality Indication (LQI) metric compatible with the Silicon Labs Zigbee stack.

Parameters

[in]	origLqi	The original LQI, for example from RAIL_RxPacketDetails_t::lqi . This parameter is not currently used but may be used in the future.
[in]	rssiDbm	The RSSI in dBm, for example from RAIL_RxPacketDetails_t::rssi .

Returns

- An LQI value (range 0..255 but not all intermediate values are possible) based on the rssiDbm and the chip's RSSI sensitivity range.

This function is compatible with [RAIL_ConvertLqiCallback_t](#) and is suitable to pass to [RAIL_ConvertLqi\(\)](#).

Definition at line 1538 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ConvertRssiToEd

uint8_t RAIL_IEEE802154_ConvertRssiToEd (int8_t rssiDbm)

Convert RSSI into 802.15.4 Energy Detection (ED) metric compatible with the Silicon Labs Zigbee stack.

Parameters

[in]	rssiDbm	The RSSI in dBm, for example from RAIL_RxPacketDetails_t::rssi .

Returns

- An Energy Detect value (range 0..255 but not all intermediate values are possible) based on the rssiDbm and the chip's RSSI sensitivity range.

Definition at line 1550 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ConfigSignalIdentifier

RAIL_Status_t RAIL_IEEE802154_ConfigSignalIdentifier ([RAIL_Handle_t](#) railHandle, [RAIL_IEEE802154_SignalIdentifierMode_t](#) signalIdentifierMode)

Configure signal identifier for 802.15.4 signal detection.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	signalIdentifierMode	Mode of signal identifier operation.

This features allows detection of 2.4 GHz 802.15.4 signal on air. This function must be called once before [RAIL_IEEE802154_EnableSignalDetection\(\)](#) to configure and enable signal identifier.

To enable event for signal detection [RAIL_ConfigEvents\(\)](#) must be called for enabling [RAIL_EVENT_SIGNAL_DETECTED](#).

This function is only supported by chips where [RAIL_IEEE802154_SUPPORTS_SIGNAL_IDENTIFIER](#) and [RAIL_IEEE802154_SupportsSignalIdentifier\(\)](#) are true.

Returns

- Status code indicating success of the function call.

Definition at line 1642 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_EnableSignalDetection

RAIL_Status_t RAIL_IEEE802154_EnableSignalDetection ([RAIL_Handle_t](#) railHandle, bool enable)

Enable or disable signal identifier for 802.15.4 signal detection.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	Signal detection is enabled if true, disabled if false.

[RAIL_IEEE802154_ConfigSignalIdentifier\(\)](#) must be called once before calling this function to configure and enable signal identifier. Once a signal is detected signal detection will be turned off and this function should be called to re-enable the signal detection without needing to call [RAIL_IEEE802154_ConfigSignalIdentifier\(\)](#) if the signal identifier is already configured and enabled.

This function is only supported by chips where [RAIL_IEEE802154_SUPPORTS_SIGNAL_IDENTIFIER](#) and [RAIL_IEEE802154_SupportsSignalIdentifier\(\)](#) are true.

Returns

- Status code indicating success of the function call.

Definition at line 1664 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ConfigCcaMode

RAIL_Status_t RAIL_IEEE802154_ConfigCcaMode ([RAIL_Handle_t](#) railHandle, [RAIL_IEEE802154_CcaMode_t](#) ccaMode)

Set 802.15.4 CCA mode.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	ccaMode	Mode of CCA operation.

This function sets the CCA mode [RAIL_IEEE802154_CcaMode_t](#). If not called, [RAIL_IEEE802154_CCA_MODE_RSSI](#) (RSSI-based CCA) is used for CCA.

In [RAIL_IEEE802154_CCA_MODE_SIGNAL](#), [RAIL_IEEE802154_CCA_MODE_SIGNAL_OR_RSSI](#) and [RAIL_IEEE802154_CCA_MODE_SIGNAL_AND_RSSI](#) the signal identifier is enabled for the duration of LBT. If previously enabled by [RAIL_IEEE802154_ConfigSignalIdentifier\(\)](#), the signal identifier will remain active until triggered.

This function is only supported by chips where [RAIL_IEEE802154_SUPPORTS_SIGNAL_IDENTIFIER](#) and [RAIL_IEEE802154_SupportsSignalIdentifier\(\)](#) are true.

Returns

-

Status code indicating success of the function call. An error should be returned if ccaMode is unsupported on a given device.

Definition at line 1695 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ConfigRxChannelSwitching

RAIL_Status_t RAIL_IEEE802154_ConfigRxChannelSwitching (RAIL_Handle_t railHandle, const RAIL_IEEE802154_RxChannelSwitchingCfg_t * pConfig)

Configure RX channel switching for 802.15.4.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	pConfig	A pointer to RAIL_IEEE802154_RxChannelSwitchingCfg_t structure. NULL will disable any switching previously set up. The feature can also be disabled by disabling RAIL_RX_OPTION_CHANNEL_SWITCHING.

Returns

- Status code indicating success of the function call. For series 3 parts: return RAIL_STATUS_INVALID_STATE if IEEE 802.15.4 is not enabled via RAIL_IEEE802154_Init(), return RAIL_STATUS_NO_ERROR otherwise.

This function configures RX channel switching, allowing reception of 2.4 GHz 802.15.4 signals on two different radio channels within the same PHY. This function should be called once before RAIL_StartRx() and/or enabling RAIL_RX_OPTION_CHANNEL_SWITCHING.

When RAIL_RX_OPTION_CHANNEL_SWITCHING is enabled, channel switching will occur during normal listening but is suspended when starting any kind of transmit, including scheduled or CSMA transmits. It remains suspended after a RAIL_TX_OPTION_WAIT_FOR_ACK transmit until the Ack is received or times out.

When RAIL_RX_OPTION_CHANNEL_SWITCHING is disabled after switching has been active, the radio could be left listening on either channel, so the application should call RAIL_StartRx() to put it on the desired non-switching channel.

Note

- Switching is cancelled on any PHY change, so this function would need to be re-called to reestablish switching after such a change. When RX channel switching is active, receive sensitivity and performance are slightly impacted.
- Switching is stopped on any channel change that is not from RAIL_IEEE802154_RxChannelSwitchingCfg_t but is resumed on configuring back the appropriate channel.

Note

- For series 2 parts: If the two channels are same, the function behaves the same as if pConfig was NULL.
- IEEE 802.15.4 must be enabled via RAIL_IEEE802154_Init(), and the radio must be in the idle state when configuring RX channel switching. A DMA channel must be allocated with RAIL_UseDma() or by incorporating the Direct Memory Access (DMA) Utility plugin; otherwise this API will return RAIL_STATUS_INVALID_CALL. This feature also requires a PRS channel, internally allocated by the RAIL library, to use and hold onto for future use. If no PRS channel is available, the function returns RAIL_STATUS_INVALID_PARAMETER.
- This function internally uses RAIL_EnableCacheSynthCal() to enable/disable the sequencer cache to store the synth calibration value.

Warnings

- For series 2 parts, this function relies on LDMA access and RAIL is meant to run in TrustZone non-secure world, it is not supported if LDMA is configured as secure peripheral and it will return RAIL_STATUS_INVALID_CALL.

Note

- For series 3 parts:
- IEEE 802.15.4 must be enabled via RAIL_IEEE802154_Init().

If the two channels are same, the function will do channel switching on a single frequency but will still duty cycle if PHY allows it.

- [RAIL_RX_OPTION_CHANNEL_SWITCHING](#) is only considered when [RAIL_StartRx\(\)](#) or [RAIL_PrepareChannel\(\)](#) is called, not instantly.

Definition at line 1787 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Address_t

Representation of 802.15.4 address.

This structure is only used for received source address information needed to perform Frame Pending lookup.

Public Attributes

uint16_t	shortAddress Present for 2 byte addresses.
uint8_t	longAddress Present for 8 byte addresses.
union RAIL_IEEE802154 _Address_t::@3	@4 Convenient storage for different address types.
RAIL_IEEE802154 _AddressLength_t	length Enumeration of the received address length.
RAIL_AddrFilterMa sk_t	filterMask A bitmask representing which address filter(s) this packet has passed.

Public Attribute Documentation

shortAddress

```
uint16_t RAIL_IEEE802154_Address_t::shortAddress
```

Present for 2 byte addresses.

Definition at line 209 of file protocol/ieee802154/rail_ieee802154.h

longAddress

```
uint8_t RAIL_IEEE802154_Address_t::longAddress[8]
```

Present for 8 byte addresses.

Definition at line 211 of file protocol/ieee802154/rail_ieee802154.h

@4

```
union RAIL_IEEE802154_Address_t::@3 RAIL_IEEE802154_Address_t::@4
```

Convenient storage for different address types.

Definition at line 212 of file protocol/ieee802154/rail_ieee802154.h

length

RAIL_IEEE802154_AddressLength_t RAIL_IEEE802154_Address_t::length

Enumeration of the received address length.

Definition at line 216 of file protocol/ieee802154/rail_ieee802154.h

filterMask

RAIL_AddrFilterMask_t RAIL_IEEE802154_Address_t::filterMask

A bitmask representing which address filter(s) this packet has passed.

It is undefined on platforms lacking [RAIL_SUPPORTS_ADDR_FILTER_MASK](#).

Definition at line 221 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_AddrConfig_t

A configuration structure for IEEE 802.15.4 Address Filtering.

This structure allows configuration of multi-PAN functionality by specifying multiple PAN Ids and short addresses. A packet will be received if it matches an address and its corresponding PAN Id. Long address 0 and short address 0 match against PAN Id 0, etc. The broadcast PAN Id and address will work with any address or PAN Id, respectively.

Note

- The broadcast addresses are handled separately and do not need to be specified here. Any address to be ignored should be set with all bits high.

Public Attributes

uint16_t	panId	PAN Ids for destination filtering.
uint16_t	shortAddr	A short network addresses for destination filtering.
uint8_t	longAddr	A 64-bit address for destination filtering.

Public Attribute Documentation

panId

```
uint16_t RAIL_IEEE802154_AddrConfig_t::panId[(3U)]
```

PAN Ids for destination filtering.

All must be specified. To disable a PAN Id, set it to the broadcast value, 0xFFFF.

Definition at line 245 of file `protocol/ieee802154/rail_ieee802154.h`

shortAddr

```
uint16_t RAIL_IEEE802154_AddrConfig_t::shortAddr[(3U)]
```

A short network addresses for destination filtering.

All must be specified. To disable a short address, set it to the broadcast value, 0xFFFF.

Definition at line 250 of file `protocol/ieee802154/rail_ieee802154.h`

longAddr

```
uint8_t RAIL_IEEE802154_AddrConfig_t::longAddr[(3U)][8]
```

A 64-bit address for destination filtering.

All must be specified. This field is parsed in over-the-air byte order. To disable a long address, set it to the reserved value of 0x00 00 00 00 00 00 00 00.

Definition at line 256 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Config_t

A configuration structure for IEEE 802.15.4 in RAIL.

Public Attributes

<code>const RAIL_IEEE802154_AddrConfig_t *</code>	<code>addresses</code> Configure the RAIL Address Filter to allow the given destination addresses.
<code>RAIL_AutoAckConfig_t</code>	<code>ackConfig</code> Define the Acking configuration for the IEEE 802.15.4 implementation.
<code>RAIL_StateTiming_t</code>	<code>timings</code> Define state timings for the IEEE 802.15.4 implementation.
<code>uint8_t</code>	<code>framesMask</code> Set which 802.15.4 frame types will be received, of Beacon, Data, Ack, and Command.
<code>bool</code>	<code>promiscuousMode</code> Enable promiscuous mode during configuration.
<code>bool</code>	<code>isPanCoordinator</code> Set whether the device is a PAN Coordinator during configuration.
<code>bool</code>	<code>defaultFramePendingInOutgoingAcks</code> The default value for the Frame Pending bit in outgoing Acks for packets that triggered the <code>RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND</code> event.

Public Attribute Documentation

addresses

```
const RAIL_IEEE802154_AddrConfig_t* RAIL_IEEE802154_Config_t::addresses
```

Configure the RAIL Address Filter to allow the given destination addresses.

If this pointer is NULL, defer destination address configuration. If a member of addresses is NULL, defer configuration of just that member. This can be overridden via [RAIL_IEEE802154_SetAddresses\(\)](#), or the individual members can be changed via [RAIL_IEEE802154_SetPanId\(\)](#), [RAIL_IEEE802154_SetShortAddress\(\)](#), and [RAIL_IEEE802154_SetLongAddress\(\)](#).

Definition at line 272 of file `protocol/ieee802154/rail_ieee802154.h`

ackConfig

```
RAIL_AutoAckConfig_t RAIL_IEEE802154_Config_t::ackConfig
```

Define the Acking configuration for the IEEE 802.15.4 implementation.

Definition at line 276 of file `protocol/ieee802154/rail_ieee802154.h`

timings

```
RAIL_StateTiming_t RAIL_IEEE802154_Config_t::timings
```

Define state timings for the IEEE 802.15.4 implementation.

Definition at line 280 of file `protocol/ieee802154/rail_ieee802154.h`

framesMask

```
uint8_t RAIL_IEEE802154_Config_t::framesMask
```

Set which 802.15.4 frame types will be received, of Beacon, Data, Ack, and Command.

This setting can be overridden via [RAIL_IEEE802154_AcceptFrames\(\)](#).

Definition at line 285 of file `protocol/ieee802154/rail_ieee802154.h`

promiscuousMode

```
bool RAIL_IEEE802154_Config_t::promiscuousMode
```

Enable promiscuous mode during configuration.

This can be overridden via [RAIL_IEEE802154_SetPromiscuousMode\(\)](#) afterwards.

Definition at line 290 of file `protocol/ieee802154/rail_ieee802154.h`

isPanCoordinator

```
bool RAIL_IEEE802154_Config_t::isPanCoordinator
```

Set whether the device is a PAN Coordinator during configuration.

This can be overridden via [RAIL_IEEE802154_SetPanCoordinator\(\)](#) afterwards.

Definition at line 295 of file `protocol/ieee802154/rail_ieee802154.h`

defaultFramePendingInOutgoingAcks

```
bool RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks
```

The default value for the Frame Pending bit in outgoing Acks for packets that triggered the [RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#) event.

Such an Ack's Frame Pending bit can be inverted if necessary during the handling of that event by calling [RAIL_IEEE802154_ToggleFramePending\(\)](#) (formerly [RAIL_IEEE802154_SetFramePending\(\)](#)).

Definition at line 303 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_RxChannelSwitchingCfg_t

A configuration structure for RX channel switching.

Public Attributes

uint32_t *	buffer	Pointer to contiguous global read-write memory that will be used by RAIL to store channel specific settings for concurrent listening, for series 2 parts.
uint16_t	bufferBytes	For series 2 parts, this parameter must be set to the length of the buffer array, in bytes.
uint16_t	channels	Array to hold the channel numbers for RX channel switching.

Public Attribute Documentation

buffer

```
uint32_t* RAIL_IEEE802154_RxChannelSwitchingCfg_t::buffer
```

Pointer to contiguous global read-write memory that will be used by RAIL to store channel specific settings for concurrent listening, for series 2 parts.

It need not be initialized and applications should never write data anywhere in this buffer. This parameter is not used for series 3 parts.

Note

- the size of this buffer must be at least as large as the [RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_BYTES](#) and needs to be word aligned.

Definition at line 345 of file `protocol/ieee802154/rail_ieee802154.h`

bufferBytes

```
uint16_t RAIL_IEEE802154_RxChannelSwitchingCfg_t::bufferBytes
```

For series 2 parts, this parameter must be set to the length of the buffer array, in bytes.

This way, during configuration, the software can confirm it's writing within the range of the buffer. The configuration API will return an error if bufferBytes is insufficient. This parameter is not used for series 3 parts.

Definition at line 353 of file `protocol/ieee802154/rail_ieee802154.h`

channels

```
uint16_t RAIL_IEEE802154_RxChannelSwitchingCfg_t::channels[(2U)]
```

Array to hold the channel numbers for RX channel switching.

Note

- Radio will switch between the exact channels specified, and not across an inclusive range of channels between the specified channels.

Definition at line 359 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ModeSwitchPhr_t

A structure containing the PHY Mode Id value and the corresponding mode switch PHR as defined in Wi-SUN spec.

These structures are usually generated by the radio configurator.

Public Attributes

uint8_t	phyModelId
	PHY mode Id.
uint16_t	phr
	Corresponding Mode Switch PHY header.

Public Attribute Documentation

phyModelId

uint8_t RAIL_IEEE802154_ModeSwitchPhr_t::phyModelId

PHY mode Id.

Definition at line 1238 of file protocol/ieee802154/rail_ieee802154.h

phr

uint16_t RAIL_IEEE802154_ModeSwitchPhr_t::phr

Corresponding Mode Switch PHY header.

Definition at line 1240 of file protocol/ieee802154/rail_ieee802154.h

IEEE 802.15.4 Radio Configurations

IEEE 802.15.4 Radio Configurations

Radio configurations for the RAIL 802.15.4 Accelerator.

These radio configurations are used to configure 802.15.4 when a function such as [RAIL_IEEE802154_Config2p4GHzRadio\(\)](#) is called. Each radio configuration listed below is compiled into the RAIL library as a weak symbol that will take into account per-die defaults. If the board configuration in use has different settings than the default, such as a different radio subsystem clock frequency, these radio configurations can be overridden to account for those settings.

Variables

<code>const</code> <code>RAIL_ChannelConfig_t</code> *const	RAIL_IEEE802154_Phy2p4GHz Default PHY to use for 2.4 GHz 802.15.4.
<code>const</code> <code>RAIL_ChannelConfig_t</code> *const	RAIL_IEEE802154_Phy2p4GHzAntDiv Default PHY to use for 2.4 GHz 802.15.4 with antenna diversity.
<code>const</code> <code>RAIL_ChannelConfig_t</code> *const	RAIL_IEEE802154_Phy2p4GHzCoex Default PHY to use for 2.4 GHz 802.15.4 optimized for coexistence.
<code>const</code> <code>RAIL_ChannelConfig_t</code> *const	RAIL_IEEE802154_Phy2p4GHzAntDivCoex Default PHY to use for 2.4 GHz 802.15.4 optimized for coexistence, while supporting antenna diversity.
<code>const</code> <code>RAIL_ChannelConfig_t</code> *const	RAIL_IEEE802154_Phy2p4GHzFem Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module.
<code>const</code> <code>RAIL_ChannelConfig_t</code> *const	RAIL_IEEE802154_Phy2p4GHzAntDivFem Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module and antenna diversity.
<code>const</code> <code>RAIL_ChannelConfig_t</code> *const	RAIL_IEEE802154_Phy2p4GHzCoexFem Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module and is optimized for radio coexistence.
<code>const</code> <code>RAIL_ChannelConfig_t</code> *const	RAIL_IEEE802154_Phy2p4GHzAntDivCoexFem Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module and antenna diversity, and is optimized for radio coexistence.
<code>const</code> <code>RAIL_ChannelConfig_t</code> *const	RAIL_IEEE802154_Phy2p4GHzCustom1 Default PHY to use for 2.4 GHz 802.15.4 with custom settings.
<code>const</code> <code>RAIL_ChannelConfig_t</code> *const	RAIL_IEEE802154_PhyGB863MHz Default PHY to use for 863 MHz GB868 802.15.4.
<code>const</code> <code>RAIL_ChannelConfig_t</code> *const	RAIL_IEEE802154_PhyGB915MHz Default PHY to use for 915 MHz GB868 802.15.4.

const
RAIL_ChannelConfig_t
ig_t *const

RAIL_IEEE802154_Phy2p4GHzRxChSwitching
Default PHY to use for 2.4 GHz 802.15.4 with RX channel switching.

Variable Documentation

RAIL_IEEE802154_Phy2p4GHz

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHz
```

Default PHY to use for 2.4 GHz 802.15.4.

Will be NULL if [RAIL_SUPPORTS_PROTOCOL_IEEE802154](#) or [RAIL_SUPPORTS_2P4GHZ_BAND](#) is 0.

Definition at line 379 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Phy2p4GHzAntDiv

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzAntDiv
```

Default PHY to use for 2.4 GHz 802.15.4 with antenna diversity.

Will be NULL if [RAIL_SUPPORTS_PROTOCOL_IEEE802154](#), [RAIL_SUPPORTS_2P4GHZ_BAND](#), or [RAIL_SUPPORTS_ANTENNA_DIVERSITY](#) is 0.

Definition at line 400 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Phy2p4GHzCoex

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzCoex
```

Default PHY to use for 2.4 GHz 802.15.4 optimized for coexistence.

Will be NULL if [RAIL_IEEE802154_SUPPORTS_COEX_PHY](#) is 0.

Definition at line 406 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Phy2p4GHzAntDivCoex

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzAntDivCoex
```

Default PHY to use for 2.4 GHz 802.15.4 optimized for coexistence, while supporting antenna diversity.

Will be NULL if [RAIL_SUPPORTS_ANTENNA_DIVERSITY](#) or [RAIL_IEEE802154_SUPPORTS_COEX_PHY](#) is 0.

Definition at line 414 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Phy2p4GHzFem

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzFem
```

Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module.

Will be NULL if [RAIL_IEEE802154_SUPPORTS_FEM_PHY](#) is 0.

Definition at line 421 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_Phy2p4GHzAntDivFem

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzAntDivFem
```

Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module and antenna diversity.

Will be NULL if [RAIL_IEEE802154_SUPPORTS_FEM_PHY](#) or [RAIL_SUPPORTS_ANTENNA_DIVERSITY](#) is 0.

Definition at line 429 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_Phy2p4GHzCoexFem

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzCoexFem
```

Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module and is optimized for radio coexistence.

Will be NULL if [RAIL_IEEE802154_SUPPORTS_FEM_PHY](#) or [RAIL_IEEE802154_SUPPORTS_COEX_PHY](#) is 0.

Definition at line 437 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_Phy2p4GHzAntDivCoexFem

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzAntDivCoexFem
```

Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module and antenna diversity, and is optimized for radio coexistence.

Will be NULL if [RAIL_IEEE802154_SUPPORTS_FEM_PHY](#), [RAIL_IEEE802154_SUPPORTS_COEX_PHY](#), or [RAIL_SUPPORTS_ANTENNA_DIVERSITY](#) is 0.

Definition at line 446 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_Phy2p4GHzCustom1

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzCustom1
```

Default PHY to use for 2.4 GHz 802.15.4 with custom settings.

Will be NULL if [RAIL_IEEE802154_SUPPORTS_CUSTOM1_PHY](#) is 0.

Definition at line 452 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_PhyGB863MHz


```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_PhyGB863MHz
```

Default PHY to use for 863 MHz GB868 802.15.4.

Will be NULL if [RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868](#) is 0.

Definition at line 458 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_PhyGB915MHz

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_PhyGB915MHz
```

Default PHY to use for 915 MHz GB868 802.15.4.

Will be NULL if [RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868](#) is 0.

Definition at line 464 of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_Phy2p4GHzRxChSwitching

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzRxChSwitching
```

Default PHY to use for 2.4 GHz 802.15.4 with RX channel switching.

Will be NULL if [RAIL_IEEE802154_SUPPORTS_RX_CHANNEL_SWITCHING](#) is 0.

Definition at line 470 of file `protocol/ieee802154/rail_ieee802154.h`

Multi-Level Frequency Modulation

Multi-Level Frequency Modulation

MFM configuration routines.

Note

- This feature is only supported on EFR32xG23 devices.

This feature can be used to directly control the TX interpolation filter input to allow for a more flexible frequency modulation scheme than the standard MODEM. When doing this, the MFM buffer is treated as an array of 8-bit signed data used as normalized frequency deviation to the SYNTH frequency to directly control the interpolation filter input. No support for frame handling, coding, nor shaping is supported. Only compatible with FSK modulations.

The functions in this group configure RAIL Multi-Level Frequency Modulation (MFM) hardware acceleration features.

To configure MFM functionality, the application must first set up a RAIL instance with [RAIL_Init\(\)](#) and other setup functions. Before enabling MFM, a ping-pong buffer (called `buffer0` and `buffer1` below) must be configured via [RAIL_SetMfmPingPongFifo\(\)](#) and populated with the initial buffer content. MFM is enabled by setting [RAIL_TxDataSource_t::TX_MFM_DATA](#) using [RAIL_ConfigData\(\)](#) and is activated when transmit is started by [RAIL_StartTx\(\)](#). Once transmitting the data in the ping-pong buffers, RAIL will manage them so it looks like a continuous transmission to the receiver. Every time one of the ping-pong buffers has been transmitted, [RAIL_EVENT_MFM_TX_BUFFER_DONE](#) is triggered so the application can update the data in that buffer without the need to start/stop the transmission. [RAIL_EVENT_MFM_TX_BUFFER_DONE](#) can be enable with [RAIL_ConfigEvents\(\)](#). Use [RAIL_StopTx\(\)](#) to finish transmitting.

```

#define MFM_RAW_BUF_WORDS 128
extern RAIL_Handle_t railHandle;
uint8_t txCount = 0;
uint32_t mfmPingPongBuffers[2][MFM_RAW_BUF_WORDS];

typedef struct mfmConfigApp {
    RAIL_MFM_PingPongBufferConfig_t buffer;
    RAIL_StateTiming_t timings;
    RAIL_DataConfig_t dataConfig;
} mfmConfigApp_t;

static mfmConfigApp_t mfmConfig = {
    .buffer = {
        .pBuffer0 = (&mfmPingPongBuffers[0]),
        .pBuffer1 = (&mfmPingPongBuffers[1]),
        .bufferSizeWords = MFM_RAW_BUF_WORDS,
    },
    .timings = {
        .idleToTx = 100,
        .idleToRx = 0,
        .rxToTx = 0,
        .txToRx = 0,
        .rxSearchTimeout = 0,
        .txToRxSearchTimeout = 0
    },
    .dataConfig = {
        .txSource = TX_MFM_DATA,
        .rxSource = RX_PACKET_DATA,
        .txMethod = PACKET_MODE,
        .rxMethod = PACKET_MODE,
    },
};

// Main RAIL events handler callback
static void RAILCb_Event(RAIL_Handle_t railHandle, RAIL_Events_t events)
{
    // Increment TX counter
    if (events & RAIL_EVENT_MFM_BUF_DONE) {
        txCount++;
        return;
    }
}

void mfmInit(void)
{
    // initialize MFM
    uint32_t idx;
    uint32_t *pDst0 = mfmConfig.buffer.pBuffer0;
    uint32_t *pDst1 = mfmConfig.buffer.pBuffer1;
    for (idx = 0; idx < (mfmConfig.buffer.bufferSizeWords / 4); idx++) {
        pDst0[4 * idx + 0] = 0x755A3100;
        pDst1[4 * idx + 0] = 0x755A3100;
        pDst0[4 * idx + 1] = 0x315A757F;
        pDst1[4 * idx + 1] = 0x315A757F;
        pDst0[4 * idx + 2] = 0x8BA6CF00;
        pDst1[4 * idx + 2] = 0x8BA6CF00;
        pDst0[4 * idx + 3] = 0xCFA68B81;
        pDst1[4 * idx + 3] = 0xCFA68B81;
    }

    RAIL_Status_t status;
    status = RAIL_SetMfmPingPongFifo(railHandle, &mfmConfig.buffer);
    assert(status == RAIL_STATUS_NO_ERROR);

    status = RAIL_SetStateTiming(railHandle, &mfmConfig.timings);
    assert(status == RAIL_STATUS_NO_ERROR);
}

```

```
mfmConfig.dataConfig.txSource = TX_MFM_DATA;
status = RAIL_ConfigData(railHandle, &mfmConfig.dataConfig); assert(status == RAIL_STATUS_NO_ERROR); // start transmitting
status = RAIL_StartTx(railHandle, 0, 0, NULL); assert(status == RAIL_STATUS_NO_ERROR);

void mfmDeInit(void){
    RAIL_Status_t status;
    status = RAIL_StopTx(railHandle, RAIL_STOP_MODES_ALL); assert(status == RAIL_STATUS_NO_ERROR);

    mfmConfig.dataConfig.txSource = TX_PACKET_DATA;
    status = RAIL_ConfigData(railHandle, &mfmConfig.dataConfig); assert(status == RAIL_STATUS_NO_ERROR);
}
```

Modules

[RAIL_MFM_PingPongBufferConfig_t](#)

Functions

[RAIL_Status_t](#) [RAIL_SetMfmPingPongFifo](#)([RAIL_Handle_t](#) railHandle, const [RAIL_MFM_PingPongBufferConfig_t](#) *config)
Set MFM ping-pong buffer.

Function Documentation

RAIL_SetMfmPingPongFifo

```
RAIL_Status_t RAIL_SetMfmPingPongFifo (RAIL\_Handle\_t railHandle, const RAIL\_MFM\_PingPongBufferConfig\_t * config)
```

Set MFM ping-pong buffer.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	config	A non-NULL pointer to the MFM ping-pong buffer configuration structure.

Returns

- Status code indicating success of the function call.

Definition at line 186 of file `common/rail_mfm.h`

RAIL_MFM_PingPongBufferConfig_t

A configuration structure for MFM Ping-pong buffer in RAIL.

Public Attributes

uint32_t *	pBuffer0	Pointer to buffer 0.
uint32_t *	pBuffer1	Pointer to buffer 1.
uint32_t	bufferSizeWords	Size of each buffer in 32-bit words.

Public Attribute Documentation

pBuffer0

```
uint32_t* RAIL_MFM_PingPongBufferConfig_t::pBuffer0
```

Pointer to buffer 0.
Must be 32-bit aligned.

Definition at line 170 of file `common/rail_mfm.h`

pBuffer1

```
uint32_t* RAIL_MFM_PingPongBufferConfig_t::pBuffer1
```

Pointer to buffer 1.
Must be 32-bit aligned.

Definition at line 172 of file `common/rail_mfm.h`

bufferSizeWords

```
uint32_t RAIL_MFM_PingPongBufferConfig_t::bufferSizeWords
```

Size of each buffer in 32-bit words.

Definition at line 174 of file `common/rail_mfm.h`

Sidewalk Radio Configurations

Sidewalk Radio Configurations

Radio configurations for the RAIL Sidewalk Accelerator.

These radio configurations are used to configure Sidewalk when a function such as `RAIL_Sidewalk_ConfigPhy2GFSK50kbps()` is called. Each radio configuration listed below is compiled into the RAIL library as a weak symbol that will take into account per-die defaults. If the board configuration in use has different settings than the default, such as a different radio subsystem clock frequency, these radio configurations can be overridden to account for those settings.

Variables

<code>const RAIL_ChannelConfig_t *const</code>	<code>RAIL_Sidewalk_Phy2GFSK50kbps</code>
	Default PHY to use for Sidewalk 2GFSK 50kbps.

Functions

<code>RAIL_Status_t</code>	<code>RAIL_Sidewalk_ConfigPhy2GFSK50kbps(RAIL_Handle_t railHandle)</code>
	Switch to the 2GFSK 50 kbps Sidewalk PHY.

Variable Documentation

RAIL_Sidewalk_Phy2GFSK50kbps

```
const RAIL_ChannelConfig_t* const RAIL_Sidewalk_Phy2GFSK50kbps
```

Default PHY to use for Sidewalk 2GFSK 50kbps.

Will be NULL if `RAIL_SUPPORTS_PROTOCOL_SIDEWALK` is 0.

Definition at line 62 of file `protocol/sidewalk/rail_sidewalk.h`

Function Documentation

RAIL_Sidewalk_ConfigPhy2GFSK50kbps

```
RAIL_Status_t RAIL_Sidewalk_ConfigPhy2GFSK50kbps (RAIL_Handle_t railHandle)
```

Switch to the 2GFSK 50 kbps Sidewalk PHY.

Parameters

[in]	<code>railHandle</code>	A handle for RAIL instance.

Returns

- Status code indicating success of the function call.

Use this function to switch to the 2GFSK 50 kbps Sidewalk PHY.

Note

- The Sidewalk PHY is supported only on some parts. The preprocessor symbol [RAIL_SUPPORTS_PROTOCOL_SIDEWALK](#) and the runtime function [RAIL_SupportsProtocolSidewalk\(\)](#) may be used to test for support of the Sidewalk PHY.

Definition at line 79 of file protocol/sidewalk/rail_sidewalk.h

Wireless M-Bus

Wireless M-Bus

Wireless M-Bus (WMBUS) configuration.

Enumerations

```
enum RAIL_WMBUS_Phy_t {
    RAIL_WMBUS_ModeTFrameA = 0U
    RAIL_WMBUS_ModeCFrameA = 2U
    RAIL_WMBUS_ModeCFrameB = 3U
}
The RX variant of the WMBUS T+C PHY.
```

Functions

```
RAIL_Status_t RAIL_WMBUS_Config(RAIL_Handle_t railHandle, bool enableSimultaneousTCRx)
Configure WMBUS simultaneous M2O RX of T and C modes feature.
```

Enumeration Documentation

RAIL_WMBUS_Phy_t

RAIL_WMBUS_Phy_t

The RX variant of the WMBUS T+C PHY.

Enumerator	
RAIL_WMBUS_ModeTFrameA	RAIL_RxPacketDetails_t::subPhyId indicating a mode T frame A packet
RAIL_WMBUS_ModeCFrameA	RAIL_RxPacketDetails_t::subPhyId indicating a mode C frame A packet
RAIL_WMBUS_ModeCFrameB	RAIL_RxPacketDetails_t::subPhyId indicating a mode C frame B packet

Definition at line 49 of file protocol/wmbus/rail_wmbus.h

Function Documentation

RAIL_WMBUS_Config

RAIL_Status_t RAIL_WMBUS_Config (RAIL_Handle_t railHandle, bool enableSimultaneousTCRx)

Configure WMBUS simultaneous M2O RX of T and C modes feature.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enableSimultaneousTCRx	true to enable WMBUS simultaneous M2O RX of T and C modes.

Returns

- Status code indicating success of the function call.

If simultaneous M2O RX of T and C modes is enabled, when PHY_wMbus_ModeTC_M2O_100k_frameA is loaded, mode T Frame A and mode C frame A/B can be received. The mode and frame type of the last received packet is available in [RAIL_RxPacketDetails_t::subPhyId](#).

Note

- This WMBUS feature is supported only on some parts. The preprocessor symbol [RAIL_WMBUS_SUPPORTS_SIMULTANEOUS_T_C_RX](#) and the runtime function [RAIL_WMBUS_SupportsSimultaneousTCRx\(\)](#) may be used to test for support.

Definition at line 84 of file `protocol/wmbus/rail_wmbus.h`

Z-Wave

Z-Wave

Z-Wave configuration routines.

The functions in this group configure RAIL Z-Wave hardware acceleration features.

To configure Z-Wave functionality, the application must first set up a RAIL instance with [RAIL_Init\(\)](#) and other setup functions.

```
RAIL_ZWAVE_NodeId_t gRecentBeamNodeId;
uint8_t gRecentBeamChannelIndex;

// Main RAIL_EVENT callback
static void RAILCb_Event(RAIL_Handle_t railHandle, RAIL_Events_t events)
{
    // Get beam Node Id and channel index from beam packet
    if (events & RAIL_EVENT_ZWAVE_BEAM) {
        if (RAIL_ZWAVE_IsEnabled(railHandle)) {
            if (RAIL_ZWAVE_GetBeamNodeId(railHandle, &gRecentBeamNodeId)
                != RAIL_STATUS_NO_ERROR)
                || (RAIL_ZWAVE_GetBeamChannelIndex(railHandle, &gRecentBeamChannelIndex)
                    != RAIL_STATUS_NO_ERROR)) {
                return;
            }
        }
    }
}

static const RAIL_ZWAVE_Config_t zwaveConfig = {
    .options = RAIL_ZWAVE_OPTIONS_DEFAULT
};

RAIL_Status_t zwaveInit(void)
{
    // initialize Z-Wave
    RAIL_Status_t status = RAIL_ZWAVE_Init(railHandle, &zwaveConfig);

    if (status != RAIL_STATUS_NO_ERROR) {
        return status;
    }

    uint8_t myHomeId[4] = { 0xDE, 0xAD, 0xBE, 0xEF };
    RAIL_ZWAVE_SetNodeId(railHandle, RAIL_ZWAVE_NODE_ID_DEFAULT);
    RAIL_ZWAVE_SetHomeId(railHandle, myHomeId, RAIL_ZWAVE_HOME_ID_HASH_DONT_CARE);

    // configure region to EU(European Union)
    return RAIL_ZWAVE_ConfigRegion(railHandle, RAIL_ZWAVE_REGION_EU);
}
```

Modules

[RAIL_ZWAVE_Config_t](#)

[RAIL_ZWAVE_LrAckData_t](#)

[RAIL_ZWAVE_BeamRxConfig_t](#)
[RAIL_ZWAVE_RegionConfig_t](#)
[RAIL_ZWAVE_IrcalVal_t](#)

Enumerations

```
enum RAIL\_ZWAVE\_Options\_t {
    RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE_SHIFT = 0
    RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES_SHIFT = 1
    RAIL_ZWAVE_OPTION_NODE_ID_FILTERING_SHIFT = 2
    RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE_SHIFT = 3
}
Z-Wave options.
```

```
enum RAIL\_ZWAVE\_NodeId\_t {
    RAIL_ZWAVE_NODE_ID_NONE = 0x00U
    RAIL_ZWAVE_NODE_ID_BROADCAST = 0xFFU
    RAIL_ZWAVE_NODE_ID_DEFAULT = RAIL_ZWAVE_NODE_ID_BROADCAST
    RAIL_ZWAVE_NODE_ID_BROADCAST_LONGRANGE = 0xFFFFU
    RAIL_ZWAVE_NODE_ID_DEFAULT_LONGRANGE = RAIL_ZWAVE_NODE_ID_BROADCAST_LONGRANGE
}
A Z-Wave Node Id.
```

```
enum RAIL\_ZWAVE\_HomId\_t {
    RAIL_ZWAVE_HOME_ID_UNKNOWN = 0x00000000U
    RAIL_ZWAVE_HOME_ID_DEFAULT = 0x54545454U
}
A Z-Wave Home Id.
```

```
enum RAIL\_ZWAVE\_HomIdHash\_t {
    RAIL_ZWAVE_HOME_ID_HASH_ILLEGAL_1 = 0x0AU
    RAIL_ZWAVE_HOME_ID_HASH_ILLEGAL_2 = 0x4AU
    RAIL_ZWAVE_HOME_ID_HASH_ILLEGAL_3 = 0x55U
    RAIL_ZWAVE_HOME_ID_HASH_DONT_CARE = 0x55U
    RAIL_ZWAVE_HOME_ID_HASH_DEFAULT = RAIL_ZWAVE_HOME_ID_HASH_DONT_CARE
}
A Z-Wave Home Id hash.
```

```
enum RAIL\_ZWAVE\_Baud\_t {
    RAIL_ZWAVE_BAUD_9600
    RAIL_ZWAVE_BAUD_40K
    RAIL_ZWAVE_BAUD_100K
    RAIL_ZWAVE_LR
    RAIL_ZWAVE_ENERGY_DETECT = RAIL_ZWAVE_LR
    RAIL_ZWAVE_BAUD_INVALID
}
Z-Wave supported baud rates or PHYs.
```

```
enum RAIL_ZWAVE_RegionId_t {  
    RAIL_ZWAVE_REGIONID_UNKNOWN = 0  
    RAIL_ZWAVE_REGIONID_EU = 1  
    RAIL_ZWAVE_REGIONID_US = 2  
    RAIL_ZWAVE_REGIONID_ANZ = 3  
    RAIL_ZWAVE_REGIONID_HK = 4  
    RAIL_ZWAVE_REGIONID_MY = 5  
    RAIL_ZWAVE_REGIONID_IN = 6  
    RAIL_ZWAVE_REGIONID_JP = 7  
    RAIL_ZWAVE_REGIONID_RU = 8  
    RAIL_ZWAVE_REGIONID_IL = 9  
    RAIL_ZWAVE_REGIONID_KR = 10  
    RAIL_ZWAVE_REGIONID_CN = 11  
    RAIL_ZWAVE_REGIONID_US_LR1 = 12  
    RAIL_ZWAVE_REGIONID_US_LR2 = 13  
    RAIL_ZWAVE_REGIONID_US_LR3 = 14  
    RAIL_ZWAVE_REGIONID_US_LR_END_DEVICE = RAIL_ZWAVE_REGIONID_US_LR3  
    RAIL_ZWAVE_REGIONID_EU_LR1 = 15  
    RAIL_ZWAVE_REGIONID_EU_LR2 = 16  
    RAIL_ZWAVE_REGIONID_EU_LR3 = 17  
    RAIL_ZWAVE_REGIONID_EU_LR_END_DEVICE = RAIL_ZWAVE_REGIONID_EU_LR3  
    RAIL_ZWAVE_REGIONID_COUNT  
}  
Z-Wave region identifications.
```

Typedefs

```
typedef RAIL_RxChannelHoppingParameters_t[(4U)]  
RAIL_RxChannelHoppingParameter_t  
Rx channel hopping on-channel time for all Z-Wave channels in a region.
```

Variables

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_EU  
EU-European Union.  
  
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_US  
US-United States.  
  
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_ANZ  
ANZ-Australia/New Zealand.  
  
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_HK  
HK-Hong Kong.  
  
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_MY  
MY-Malaysia.  
  
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_IN  
IN-India.  
  
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_JP  
JP-Japan.  
  
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_IL  
IL-Israel.
```

<code>onConfig_t</code>	<code>RAIL_ZWAVE_REGION_JPED</code> JP-Japan Energy-Detect.
<code>const</code> <code>RAIL_ZWAVE_Regi</code> <code>onConfig_t</code>	<code>RAIL_ZWAVE_REGION_RU</code> RU-Russia.
<code>const</code> <code>RAIL_ZWAVE_Regi</code> <code>onConfig_t</code>	<code>RAIL_ZWAVE_REGION_IL</code> IL-Israel.
<code>const</code> <code>RAIL_ZWAVE_Regi</code> <code>onConfig_t</code>	<code>RAIL_ZWAVE_REGION_KR</code> KR-Korea.
<code>const</code> <code>RAIL_ZWAVE_Regi</code> <code>onConfig_t</code>	<code>RAIL_ZWAVE_REGION_KRED</code> KR-Korea Energy-Detect.
<code>const</code> <code>RAIL_ZWAVE_Regi</code> <code>onConfig_t</code>	<code>RAIL_ZWAVE_REGION_CN</code> CN-China.
<code>const</code> <code>RAIL_ZWAVE_Regi</code> <code>onConfig_t</code>	<code>RAIL_ZWAVE_REGION_US_LR1</code> US-Long Range 1.
<code>const</code> <code>RAIL_ZWAVE_Regi</code> <code>onConfig_t</code>	<code>RAIL_ZWAVE_REGION_US_LR2</code> US-Long Range 2.
<code>const</code> <code>RAIL_ZWAVE_Regi</code> <code>onConfig_t</code>	<code>RAIL_ZWAVE_REGION_US_LR3</code> US-Long Range 3.
<code>const</code> <code>RAIL_ZWAVE_Regi</code> <code>onConfig_t</code>	<code>RAIL_ZWAVE_REGION_EU_LR1</code> EU-Long Range 1.
<code>const</code> <code>RAIL_ZWAVE_Regi</code> <code>onConfig_t</code>	<code>RAIL_ZWAVE_REGION_EU_LR2</code> EU-Long Range 2.
<code>const</code> <code>RAIL_ZWAVE_Regi</code> <code>onConfig_t</code>	<code>RAIL_ZWAVE_REGION_EU_LR3</code> EU-Long Range 3.
<code>const</code> <code>RAIL_ZWAVE_Regi</code> <code>onConfig_t</code>	<code>RAIL_ZWAVE_REGION_INVALID</code> Invalid Region.

Functions

<code>RAIL_Status_t</code>	<code>RAIL_ZWAVE_ConfigRegion</code> (<code>RAIL_Handle_t</code> railHandle, <code>const</code> <code>RAIL_ZWAVE_RegionConfig_t</code> *regionCfg) Switch the Z-Wave region.
<code>RAIL_Status_t</code>	<code>RAIL_ZWAVE_PerformIrcal</code> (<code>RAIL_Handle_t</code> railHandle, <code>RAIL_ZWAVE_IrcalVal_t</code> *plrCalVals, <code>bool</code> forceIrcal) Perform image rejection calibration on all valid channels of a Z-Wave region.
<code>RAIL_Status_t</code>	<code>RAIL_ZWAVE_Init</code> (<code>RAIL_Handle_t</code> railHandle, <code>const</code> <code>RAIL_ZWAVE_Config_t</code> *config) Initialize RAIL for Z-Wave features.
<code>RAIL_Status_t</code>	<code>RAIL_ZWAVE_Deinit</code> (<code>RAIL_Handle_t</code> railHandle) De-initialize Z-Wave hardware acceleration.

bool	RAIL_ZWAVE_IsEnabled (RAIL_Handle_t railHandle) Return whether Z-Wave hardware acceleration is currently enabled.
RAIL_Status_t	RAIL_ZWAVE_ConfigOptions (RAIL_Handle_t railHandle, RAIL_ZWAVE_Options_t mask, RAIL_ZWAVE_Options_t options) Configure Z-Wave options.
RAIL_Status_t	RAIL_ZWAVE_SetNodeId (RAIL_Handle_t railHandle, RAIL_ZWAVE_NodeId_t nodeId) Inform RAIL of the Z-Wave node's Node Id for receive filtering.
RAIL_Status_t	RAIL_ZWAVE_SetHomeId (RAIL_Handle_t railHandle, RAIL_ZWAVE_HomeId_t homeId, RAIL_ZWAVE_HomeIdHash_t homeIdHash) Inform RAIL of the Z-Wave node's Home Id and its hash for receive filtering.
RAIL_Status_t	RAIL_ZWAVE_GetBeamNodeId (RAIL_Handle_t railHandle, RAIL_ZWAVE_NodeId_t *pNodeId) Get the Node Id of the most recently seen beam frame that triggered RAIL_EVENT_ZWAVE_BEAM .
RAIL_Status_t	RAIL_ZWAVE_GetBeamHomeIdHash (RAIL_Handle_t railHandle, RAIL_ZWAVE_HomeIdHash_t *pBeamHomeIdHash) Get the Home Id hash of the most recently seen beam frame that triggered RAIL_EVENT_ZWAVE_BEAM .
RAIL_Status_t	RAIL_ZWAVE_GetBeamChannelIndex (RAIL_Handle_t railHandle, uint8_t *pChannelIndex) Get the channel hopping index of the most recently seen beam frame that triggered RAIL_EVENT_ZWAVE_BEAM .
RAIL_Status_t	RAIL_ZWAVE_GetLrBeamTxPower (RAIL_Handle_t railHandle, uint8_t *pLrBeamTxPower) Get the TX power used by the transmitter of the most recently seen long range beam frame that triggered RAIL_EVENT_ZWAVE_BEAM .
RAIL_Status_t	RAIL_ZWAVE_GetBeamRssi (RAIL_Handle_t railHandle, int8_t *pBeamRssi) Get the RSSI of the received beam frame.
RAIL_Status_t	RAIL_ZWAVE_SetTxLowPower (RAIL_Handle_t railHandle, uint8_t powerLevel) Set the Raw Low Power settings.
RAIL_TxPowerLevel_t	RAIL_ZWAVE_GetTxLowPower (RAIL_Handle_t railHandle) Get the TX low power in raw units (see rail_chip_specific.h for value ranges).
RAIL_Status_t	RAIL_ZWAVE_SetTxLowPowerDbm (RAIL_Handle_t railHandle, RAIL_TxPower_t power) Set the Low Power settings in deci-dBm.
RAIL_TxPower_t	RAIL_ZWAVE_GetTxLowPowerDbm (RAIL_Handle_t railHandle) Get the TX low power in terms of deci-dBm.
RAIL_Status_t	RAIL_ZWAVE_ReceiveBeam (RAIL_Handle_t railHandle, uint8_t *beamDetectIndex, const RAIL_SchedulerInfo_t *schedulerInfo) Implement beam detection and reception algorithms.
RAIL_Status_t	RAIL_ZWAVE_ConfigBeamRx (RAIL_Handle_t railHandle, const RAIL_ZWAVE_BeamRxConfig_t *config) Configure the receive algorithm used in RAIL_ZWAVE_ReceiveBeam() .
RAIL_Status_t	RAIL_ZWAVE_SetDefaultRxBeamConfig (RAIL_Handle_t railHandle) Set the default RX beam configuration.
RAIL_Status_t	RAIL_ZWAVE_GetRxBeamConfig (RAIL_ZWAVE_BeamRxConfig_t *pConfig) Get the current RX beam configuration.
RAIL_Status_t	RAIL_ZWAVE_ConfigRxChannelHopping (RAIL_Handle_t railHandle, RAIL_RxChannelHoppingConfig_t *config) Configure the channel hop timings for use in Z-Wave RX channel hop configuration.
RAIL_ZWAVE_RegionId_t	RAIL_ZWAVE_GetRegion (RAIL_Handle_t railHandle) Get the Z-Wave region.

RAIL_Status_t

RAIL_ZWAVE_SetLrAckData(RAIL_Handle_t railHandle, const RAIL_ZWAVE_LrAckData_t *pLrAckData)
Write the Auto-Ack FIFO for the next outgoing Z-Wave Long Range Ack.

Macros

- #define RAIL_ZWAVE_OPTIONS_NONE 0U
A value representing no options.
- #define RAIL_ZWAVE_OPTIONS_DEFAULT RAIL_ZWAVE_OPTIONS_NONE
All options are disabled by default.
- #define RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE (1u << RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE_SHIFT)
An option to configure promiscuous mode, accepting non-beam packets regardless of their Home Id.
- #define RAIL_ZWAVE_OPTION_NODE_ID_FILTERING (1u << RAIL_ZWAVE_OPTION_NODE_ID_FILTERING_SHIFT)
An option to filter non-beam packets based on their Node Id when [RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE](#) is disabled.
- #define RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES (1u << RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES_SHIFT)
An option to configure beam frame recognition.
- #define RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE (1u << RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE_SHIFT)
An option to receive all beams promiscuously when [RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES](#) is enabled.
- #define RAIL_ZWAVE_OPTIONS_ALL 0xFFFFFFFFU
A value representing all options.
- #define RAIL_ZWAVE_FREQ_INVALID 0xFFFFFFFFFUL
Sentinel value to indicate that a channel (and thus its frequency) are invalid.
- #define RAIL_ZWAVE_LR_BEAM_TX_POWER_INVALID (0xFFU)
Invalid beam TX power value returned when [RAIL_ZWAVE_GetLrBeamTxPower\(\)](#) is called after receiving a regular non-long-range beam.
- #define RAIL_NUM_ZWAVE_CHANNELS (4U)
Number of channels in each of Z-Wave's region-based PHYs.
- #define RAIL_ZWAVE_REGION_US_LR_END_DEVICE RAIL_ZWAVE_REGION_US_LR3
Backwards-compatible define.
- #define RAIL_ZWAVE_REGION_EU_LR_END_DEVICE RAIL_ZWAVE_REGION_EU_LR3
Backwards-compatible define.

Enumeration Documentation

RAIL_ZWAVE_Options_t

RAIL_ZWAVE_Options_t	
Z-Wave options.	
Enumerator	
RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE_SHIFT	Shift position of RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE bit.
RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES_SHIFT	Shift position of RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES bit.

RAIL_ZWAVE_OPTION_NODE_ID_FILTERING_SHIFT	Shift position of RAIL_ZWAVE_OPTION_NODE_ID_FILTERING bit.
RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE_SHIFT	Shift position of RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE bit.

Definition at line 97 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_NodeId_t

RAIL_ZWAVE_NodeId_t

A Z-Wave Node Id.

This data type is 12 bits wide when using the ZWave Long Range PHY, and 8 bits wide otherwise.

Note

- When using the Long Range PHY, values 0xFA1..0xFFE are reserved. Otherwise, values 0xE9..0xFE are reserved.

Enumerator

RAIL_ZWAVE_NODE_ID_NONE	The unknown Node Id for uninitialized nodes.
RAIL_ZWAVE_NODE_ID_BROADCAST	The broadcast Node Id.
RAIL_ZWAVE_NODE_ID_DEFAULT	Default to the broadcast Node Id.
RAIL_ZWAVE_NODE_ID_BROADCAST_LONGRANGE	The Long Range broadcast Node Id.
RAIL_ZWAVE_NODE_ID_DEFAULT_LONGRANGE	Default to the Long Range broadcast Node Id.

Definition at line 175 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_HomeId_t

RAIL_ZWAVE_HomeId_t

A Z-Wave Home Id.

Note

- Home Ids in the range 0x54000000..0x55FFFFFF are illegal.

Enumerator

RAIL_ZWAVE_HOME_ID_UNKNOWN	The unknown Home Id.
RAIL_ZWAVE_HOME_ID_DEFAULT	An impossible and unlikely Home Id.

Definition at line 213 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_HomeIdHash_t

RAIL_ZWAVE_HomeIdHash_t

A Z-Wave Home Id hash.

Note

- Certain values (as shown) are illegal.

Enumerator

RAIL_ZWAVE_HOME_ID_HASH_ILLEGAL_1	An illegal Home Id hash value.
RAIL_ZWAVE_HOME_ID_HASH_ILLEGAL_2	An illegal Home Id hash value.
RAIL_ZWAVE_HOME_ID_HASH_ILLEGAL_3	An illegal Home Id hash value.
RAIL_ZWAVE_HOME_ID_HASH_DONT_CARE	Illegal Home Id hash value that suppresses checking the Home Id hash field of beam packets.
RAIL_ZWAVE_HOME_ID_HASH_DEFAULT	Default to don't care.

Definition at line 232 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_Baud_t

RAIL_ZWAVE_Baud_t

Z-Wave supported baud rates or PHYs.

Enumerator

RAIL_ZWAVE_BAUD_9600	9.6 kbps baud rate.
RAIL_ZWAVE_BAUD_40K	40 kbps baud rate.
RAIL_ZWAVE_BAUD_100K	100 kbps baud rate.
RAIL_ZWAVE_LR	Long Range PHY.
RAIL_ZWAVE_ENERGY_DETECT	Energy detection PHY.
RAIL_ZWAVE_BAUD_INVALID	Sentinel value for invalid baud rate.

Definition at line 280 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_RegionId_t

RAIL_ZWAVE_RegionId_t

Z-Wave region identifications.

Enumerator

RAIL_ZWAVE_REGIONID_UNKNOWN	Unknown/Invalid.
RAIL_ZWAVE_REGIONID_EU	European Union.
RAIL_ZWAVE_REGIONID_US	United States.
RAIL_ZWAVE_REGIONID_ANZ	Australia/New Zealand.
RAIL_ZWAVE_REGIONID_HK	Hong Kong.
RAIL_ZWAVE_REGIONID_MY	Malaysia.
RAIL_ZWAVE_REGIONID_IN	India.
RAIL_ZWAVE_REGIONID_JP	Japan.
RAIL_ZWAVE_REGIONID_RU	Russian Federation.
RAIL_ZWAVE_REGIONID_IL	Israel.
RAIL_ZWAVE_REGIONID_KR	Korea.
RAIL_ZWAVE_REGIONID_CN	China.
RAIL_ZWAVE_REGIONID_US_LR1	United States, with first long range PHY.

RAIL_ZWAVE_REGIONID_US_LR2	United States, with second long range PHY.
RAIL_ZWAVE_REGIONID_US_LR3	United States, with third long range PHY.
RAIL_ZWAVE_REGIONID_US_LR_END_DEVICE	Deprecated Backwards compatible name.
RAIL_ZWAVE_REGIONID_EU_LR1	European Union, with first long range PHY.
RAIL_ZWAVE_REGIONID_EU_LR2	European Union, with second long range PHY.
RAIL_ZWAVE_REGIONID_EU_LR3	European Union, with third long range PHY.
RAIL_ZWAVE_REGIONID_EU_LR_END_DEVICE	Deprecated Backwards compatible name.
RAIL_ZWAVE_REGIONID_COUNT	Count of known regions.

Definition at line 345 of file protocol/zwave/rail_zwave.h

Typedef Documentation

RAIL_RxChannelHoppingParameters_t

```
RAIL_RxChannelHoppingParameters_t [(4U)]
```

Rx channel hopping on-channel time for all Z-Wave channels in a region.

Definition at line 507 of file protocol/zwave/rail_zwave.h

Variable Documentation

RAIL_ZWAVE_REGION_EU

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_EU
```

EU-European Union.

Definition at line 945 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_US

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_US
```

US-United States.

Definition at line 948 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_ANZ

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_ANZ
```

ANZ-Australia/New Zealand.

Definition at line 951 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_HK

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_HK
```

HK-Hong Kong.

Definition at line 954 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_MY

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_MY
```

MY-Malaysia.

Definition at line 957 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_IN

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_IN
```

IN-India.

Definition at line 960 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_JP

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_JP
```

JP-Japan.

Definition at line 963 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_JPED

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_JPED
```

JP-Japan Energy-Detect.

Definition at line 966 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_RU

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_RU
```

RU-Russia.

Definition at line 969 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_IL

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_IL
```

IL-Israel.

Definition at line 972 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_KR

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_KR
```

KR-Korea.

Definition at line 975 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_KRED

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_KRED
```

KR-Korea Energy-Detect.

Definition at line 978 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_CN

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_CN
```

CN-China.

Definition at line 981 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_US_LR1

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_US_LR1
```

US-Long Range 1.

Definition at line 984 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_US_LR2

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_US_LR2
```

US-Long Range 2.

Definition at line 987 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_US_LR3

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_US_LR3
```

US-Long Range 3.

Definition at line 990 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_EU_LR1

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_EU_LR1
```

EU-Long Range 1.

Definition at line 995 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_EU_LR2

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_EU_LR2
```

EU-Long Range 2.

Definition at line 998 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_EU_LR3

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_EU_LR3
```

EU-Long Range 3.

Definition at line 1001 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_INVALID

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_INVALID
```

Invalid Region.

Definition at line 1006 of file protocol/zwave/rail_zwave.h

Function Documentation

RAIL_ZWAVE_ConfigRegion

```
RAIL_Status_t RAIL_ZWAVE_ConfigRegion (RAIL_Handle_t railHandle, const RAIL_ZWAVE_RegionConfig_t * regionCfg)
```

Switch the Z-Wave region.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	regionCfg	A pointer to a Z-Wave channel configuration for the selected region.

Returns

- Status code indicating success of the function call.

Note

- Setting a new Z-Wave Region will default any Low Power values to Normal Power values for the region. Z-Wave Region configuration must always be followed by a Low Power setup in case one desires to have the Low Power Acking functionality.

Definition at line 523 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_PerformIrcal

RAIL_Status_t RAIL_ZWAVE_PerformIrcal (RAIL_Handle_t railHandle, RAIL_ZWAVE_IrcalVal_t * plrCalVals, bool forceIrcal)

Perform image rejection calibration on all valid channels of a Z-Wave region.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	plrCalVals	An application-provided pointer of type RAIL_ZWAVE_IrcalVal_t. This is populated with image rejection calibration values, if not NULL or initialized with RAIL_CAL_INVALID_VALUE or if forceIrcal is true.
[in]	forceIrcal	If true, will always perform image rejection calibration and not use previously cached values.

Returns

- Status code indicating success of the function call.

Note

- This function also calibrates for beam detection and should be called before RAIL_ZWAVE_ReceiveBeam() and after the Z-Wave region has been configured via RAIL_ZWAVE_ConfigRegion(). Channel hopping must be disabled otherwise this function will return RAIL_STATUS_INVALID_CALL.

Definition at line 545 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_Init

RAIL_Status_t RAIL_ZWAVE_Init (RAIL_Handle_t railHandle, const RAIL_ZWAVE_Config_t * config)

Initialize RAIL for Z-Wave features.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	A pointer to a Z-Wave configuration structure.

Returns

- Status code indicating success of the function call.

This function is the entry point for working with Z-Wave within RAIL. It sets up relevant hardware acceleration for Z-Wave-specific features, such as Home Id filtering and beam packets (as specified in the configuration) and allows users to select the relevant Z-Wave region-specific PHY via [RAIL_ZWAVE_ConfigRegion\(\)](#).

Definition at line 562 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_Deinit

RAIL_Status_t RAIL_ZWAVE_Deinit ([RAIL_Handle_t](#) railHandle)

De-initialize Z-Wave hardware acceleration.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

Disables and resets all Z-Wave hardware acceleration features. This function should only be called when the radio is idle.

Definition at line 574 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_IsEnabled

bool RAIL_ZWAVE_IsEnabled ([RAIL_Handle_t](#) railHandle)

Return whether Z-Wave hardware acceleration is currently enabled.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if Z-Wave hardware acceleration was enabled to start with and false otherwise.

Definition at line 583 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_ConfigOptions

RAIL_Status_t RAIL_ZWAVE_ConfigOptions ([RAIL_Handle_t](#) railHandle, [RAIL_ZWAVE_Options_t](#) mask, [RAIL_ZWAVE_Options_t](#) options)

Configure Z-Wave options.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mask	A bitmask containing which options should be modified.
[in]	options	A bitmask containing desired configuration settings. Bit positions for each option are found in the RAIL_ZWAVE_Options_t .

Returns

- Status code indicating success of the function call.

Definition at line 594 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_SetNodeId

```
RAIL_Status_t RAIL_ZWAVE_SetNodeId (RAIL_Handle_t railHandle, RAIL_ZWAVE_NodeId_t nodeId)
```

Inform RAIL of the Z-Wave node's Node Id for receive filtering.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	nodeId	A Z-Wave Node Id.

Returns

- Status code indicating success of the function call.

Note

- Until this API is called, RAIL will assume the Node Id is [RAIL_ZWAVE_NODE_ID_DEFAULT](#).

Definition at line 608 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_SetHomeId

```
RAIL_Status_t RAIL_ZWAVE_SetHomeId (RAIL_Handle_t railHandle, RAIL_ZWAVE_HomeId_t homeId, RAIL_ZWAVE_HomeIdHash_t homeIdHash)
```

Inform RAIL of the Z-Wave node's Home Id and its hash for receive filtering.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	homeId	A Z-Wave Home Id.
[in]	homeIdHash	The hash of the Home Id expected in beam frames. If this is RAIL_ZWAVE_HOME_ID_HASH_DONT_CARE , beam frame detection will not check the Home Id hash in a received beam frame at all, and RAIL_EVENT_ZWAVE_BEAM will trigger based solely on the Node Id in the beam frame.

Returns

- Status code indicating success of the function call.

Note

- Until this API is called, RAIL will assume the Home Id is an illegal one of [RAIL_ZWAVE_HOME_ID_DEFAULT](#), and its hash is [RAIL_ZWAVE_HOME_ID_HASH_DONT_CARE](#).

Definition at line 627 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_GetBeamNodeId

```
RAIL_Status_t RAIL_ZWAVE_GetBeamNodeId (RAIL_Handle_t railHandle, RAIL_ZWAVE_NodeId_t * pNodeId)
```


Get the Node Id of the most recently seen beam frame that triggered [RAIL_EVENT_ZWAVE_BEAM](#).

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	pNodeId	A pointer to RAIL_ZWAVE_NodeId_t to populate.

Returns

- Status code indicating success of the function call.

Note

- This is best called while handling the [RAIL_EVENT_ZWAVE_BEAM](#) event; if multiple beams are received only the most recent beam's NodeId is provided.

Definition at line 645 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_GetBeamHomeIdHash

RAIL_Status_t RAIL_ZWAVE_GetBeamHomeIdHash ([RAIL_Handle_t](#) railHandle, [RAIL_ZWAVE_HomeIdHash_t](#) * pBeamHomeIdHash)

Get the Home Id hash of the most recently seen beam frame that triggered [RAIL_EVENT_ZWAVE_BEAM](#).

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	pBeamHomeIdHash	A pointer to RAIL_ZWAVE_HomeIdHash_t to populate.

Returns

- Status code indicating success of the function call.

Note

- This is best called while handling the [RAIL_EVENT_ZWAVE_BEAM](#) event; if multiple beams are received only the most recent beam's Home Id hash is provided.

Definition at line 660 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_GetBeamChannelIndex

RAIL_Status_t RAIL_ZWAVE_GetBeamChannelIndex ([RAIL_Handle_t](#) railHandle, uint8_t * pChannelIndex)

Get the channel hopping index of the most recently seen beam frame that triggered [RAIL_EVENT_ZWAVE_BEAM](#).

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	pChannelIndex	A pointer to a uint8_t to populate with the channel hopping index. If channel-hopping was off at the time the beam packet was received, RAIL_CHANNEL_HOPPING_INVALID_INDEX is provided.

Returns

- Status code indicating success of the function call.

Note

This is best called while handling the [RAIL_EVENT_ZWAVE_BEAM](#) event; if multiple beams are received only the most recent beam's channel hopping index is provided.

Definition at line 678 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_GetLrBeamTxPower

RAIL_Status_t RAIL_ZWAVE_GetLrBeamTxPower ([RAIL_Handle_t](#) railHandle, uint8_t * pLrBeamTxPower)

Get the TX power used by the transmitter of the most recently seen long range beam frame that triggered [RAIL_EVENT_ZWAVE_BEAM](#).

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	pLrBeamTxPower	An application provided pointer to a uint8_t to be populated with the TX power of the latest long range beam. This will be set to RAIL_ZWAVE_LR_BEAM_TX_POWER_INVALID if this API is called after receiving a regular non-long-range beam.

Returns

- Status code indicating success of the function call. This function will return [RAIL_STATUS_INVALID_STATE](#) if called after receiving a regular (non-long-range) beam.

Note

- This is best called while handling the [RAIL_EVENT_ZWAVE_BEAM](#) event; if multiple beams are received only the most recent long range beam's TX power is provided.
- The following table shows long range beam TX power value to dBm value mapping:

Tx Power Value	Description
0	-6 dBm
1	-2 dBm
2	+2 dBm
3	+6 dBm
4	+10 dBm
5	+13 dBm
6	+16 dBm
7	+19 dBm
8	+21 dBm
9	+23 dBm
10	+25 dBm
11	+26 dBm
12	+27 dBm
13	+28 dBm
14	+29 dBm
15	+30 dBm

Definition at line 721 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_GetBeamRssi

RAIL_Status_t RAIL_ZWAVE_GetBeamRssi ([RAIL_Handle_t](#) railHandle, int8_t * pBeamRssi)

Get the RSSI of the received beam frame.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	pBeamRssi	An application provided pointer to a int8_t to be populated with the latest beam's RSSI, in dBm.

Returns

- Status code indicating success of the function call. This function will return [RAIL_STATUS_INVALID_STATE](#) if called without ever having received a beam.

Note

- This is best called while handling the [RAIL_EVENT_ZWAVE_BEAM](#) event; if multiple beams are received only the most recent beam's RSSI is provided.

Definition at line 738 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_SetTxLowPower

RAIL_Status_t RAIL_ZWAVE_SetTxLowPower ([RAIL_Handle_t](#) railHandle, uint8_t powerLevel)

Set the Raw Low Power settings.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	powerLevel	Desired low power raw level.

Returns

- Status code indicating success of the function call.

Low Power settings are required during Ack transmissions when the Low Power Bit is set. This setting is only valid for one subsequent transmission, after which all transmissions will be at the nominal power setting, until re-invoked.

Definition at line 753 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_GetTxLowPower

RAIL_TxPowerLevel_t RAIL_ZWAVE_GetTxLowPower ([RAIL_Handle_t](#) railHandle)

Get the TX low power in raw units (see [rail_chip_specific.h](#) for value ranges).

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The chip-specific [RAIL_TxPowerLevel_t](#) raw value of the low transmit power.

This API returns the low raw power value that was set by [RAIL_ZWAVE_SetTxLowPower\(\)](#).

Calling this function before configuring the Low Power PA (i.e., before a successful call to [RAIL_ZWAVE_SetTxLowPowerDbm\(\)](#) or [RAIL_ZWAVE_SetTxLowPower\(\)](#)) will return a low power value that is the same as the nominal power. Also, calling this function before configuring the PA (i.e., before a successful call to [RAIL_ConfigTxPower\(\)](#)) will return [RAIL_TX_POWER_LEVEL_INVALID](#).

Definition at line 775 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_SetTxLowPowerDbm

`RAIL_Status_t RAIL_ZWAVE_SetTxLowPowerDbm (RAIL_Handle_t railHandle, RAIL_TxPower_t power)`

Set the Low Power settings in deci-dBm.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	power	Desired low power in deci-dBm.

Returns

- Status code indicating success of the function call.

Low Power settings are required during Ack transmissions when the Low Power Bit is set. This setting is only valid for one subsequent transmission, after which all transmissions will be at the nominal power setting, until re-invoked.

Definition at line 791 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_GetTxLowPowerDbm

`RAIL_TxPower_t RAIL_ZWAVE_GetTxLowPowerDbm (RAIL_Handle_t railHandle)`

Get the TX low power in terms of deci-dBm.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The chip-specific [RAIL_TxPower_t](#) value of the low transmit power in deci-dBm.

Definition at line 801 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_ReceiveBeam

`RAIL_Status_t RAIL_ZWAVE_ReceiveBeam (RAIL_Handle_t railHandle, uint8_t * beamDetectIndex, const RAIL_SchedulerInfo_t * schedulerInfo)`

Implement beam detection and reception algorithms.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	beamDetectIndex	A pointer to an indicator of whether or not a beam was detected at all, regardless of if it was received, generally for use only by instruction from Silicon Labs. Can be NULL.

[in]	schedulerInfo	A pointer to information to allow the radio scheduler to place this operation appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions. Note that Z-Wave currently does not support multiprotocol, so this scheduler info exists to future proof the API for when it does.

Returns

- Status code indicating success of the function call. Reasons for failure include an un-idled radio or a non-Japan non-Korea region configured before calling this function.

This function takes care of all configuration and radio setup to detect and receive beams in the current Z-Wave region. If a beam is detected, RAIL will provide the usual [RAIL_EVENT_ZWAVE_BEAM](#) event during which time users can process the beam as expected. However, normal packets may also be received during this time (also triggering [RAIL_EVENTS_RX_COMPLETION](#) events), in which case, this API may need to be re-called to receive a beam. Users should also listen for [RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE](#), which will indicate that no beam is heard. At that point, the radio will be automatically idled. Until one of these events is received, users should not try to reconfigure radio settings or start another radio operation. If an application needs to do some other operation or configuration, it must first call [RAIL_Idle\(\)](#) and wait for the radio to idle.

Note

- : The radio must be idle before calling this function.
- : [RAIL_ConfigRxChannelHopping\(\)](#) must have been called successfully in Z-Wave before this function is called to provide a valid memory buffer for internal use (see [RAIL_RxChannelHoppingConfig_t::buffer](#)).
- : This function alters radio functionality substantially. After calling it, the user should call [RAIL_ZWAVE_ConfigRegion\(\)](#), [RAIL_ConfigRxChannelHopping\(\)](#), [RAIL_EnableRxChannelHopping\(\)](#), and [RAIL_SetRxTransitions\(\)](#) to reset these parameters to whatever behaviors were desired before calling this function. Additionally, this function will idle the radio upon on exit.

Definition at line 847 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_ConfigBeamRx

```
RAIL_Status_t RAIL_ZWAVE_ConfigBeamRx (RAIL_Handle_t railHandle, const RAIL_ZWAVE_BeamRxConfig_t * config)
```

Configure the receive algorithm used in [RAIL_ZWAVE_ReceiveBeam\(\)](#).

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	A pointer to a configuration for the beam detection algorithm.

Returns

- Status code indicating success of the function call.

Warnings

- This function should not be used without direct instruction by Silicon Labs.

Definition at line 860 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_SetDefaultRxBeamConfig

```
RAIL_Status_t RAIL_ZWAVE_SetDefaultRxBeamConfig (RAIL_Handle_t railHandle)
```

Set the default RX beam configuration.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

Note

- This function resets any changes made to the beam configuration via [RAIL_ZWAVE_ConfigBeamRx\(\)](#) and the default beam configuration will be in effect on subsequent call(s) to [RAIL_ZWAVE_ReceiveBeam\(\)](#).

Definition at line 873 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_GetRxBeamConfig

```
RAIL_Status_t RAIL_ZWAVE_GetRxBeamConfig (RAIL\_ZWAVE\_BeamRxConfig\_t * pConfig)
```

Get the current RX beam configuration.

Parameters

[out]	pConfig	A pointer to RAIL_ZWAVE_BeamRxConfig_t to be populated with the current beam configuration.

Returns

- Status code indicating success of the function call.

Definition at line 884 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_ConfigRxChannelHopping

```
RAIL_Status_t RAIL_ZWAVE_ConfigRxChannelHopping (RAIL\_Handle\_t railHandle, RAIL\_RxChannelHoppingConfig\_t * config)
```

Configure the channel hop timings for use in Z-Wave RX channel hop configuration.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	config	A pointer to a configuration for Z-Wave RX channel hopping. This structure must be allocated in application global read-write memory. RAIL will populate fields within or referenced by this structure during its operation. Be sure to allocate RAIL_RxChannelHoppingConfigEntry_t entries[] for RAIL_NUM_ZWAVE_CHANNELS . Be sure to set RAIL_RxChannelHoppingConfig_t::numberOfChannels to the desired number of channels.

Returns

- Status code indicating success of the function call.

Warnings

- This function should not be used without direct instruction by Silicon Labs.

Note

- : This API must be called before [RAIL_EnableRxChannelHopping\(\)](#). This API must never be called while the radio is on with RX Duty Cycle or Channel Hopping enabled.

Definition at line 907 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_GetRegion

```
RAIL_ZWAVE_RegionId_t RAIL_ZWAVE_GetRegion (RAIL_Handle_t railHandle)
```

Get the Z-Wave region.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The [RAIL_ZWAVE_RegionId_t](#) value.

Note

- [RAIL_ZWAVE_ConfigRegion\(\)](#) must have been called successfully before this function is called. Otherwise, [RAIL_ZWAVE_REGIONID_UNKNOWN](#) is returned.

Definition at line 920 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_SetLrAckData

```
RAIL_Status_t RAIL_ZWAVE_SetLrAckData (RAIL_Handle_t railHandle, const RAIL_ZWAVE_LrAckData_t * pLrAckData)
```

Write the Auto-Ack FIFO for the next outgoing Z-Wave Long Range Ack.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	pLrAckData	An application provided pointer to a const RAIL_ZWAVE_LrAckData_t to populate the noise floor, TX power and receive rssi bytes of the outgoing Z-Wave Long Range Ack packet.

Returns

- Status code indicating success of the function call.

This function sets the Auto-Ack data to use in acknowledging the frame being received. It must only be called while processing the [RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND](#). This will return [RAIL_STATUS_INVALID_STATE](#) if it is too late to write the outgoing Ack. When successful, the ackData will only be sent once. Subsequent packets needing an Z-Wave Long Range Ack will each need to call this function to write the Ack information.

Definition at line 939 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_Config_t

A configuration structure for Z-Wave in RAIL.

Public Attributes

RAIL_ZWAVE_Options_t	options Defines Z-Wave options.
RAIL_AutoAckConfig_t	ackConfig Defines Z-Wave Acking configuration.
RAIL_StateTiming_t	timings Defines state timings for Z-Wave.

Public Attribute Documentation

options

RAIL_ZWAVE_Options_t RAIL_ZWAVE_Config_t::options

Defines Z-Wave options.

Definition at line 265 of file `protocol/zwave/rail_zwave.h`

ackConfig

RAIL_AutoAckConfig_t RAIL_ZWAVE_Config_t::ackConfig

Defines Z-Wave Acking configuration.

Definition at line 269 of file `protocol/zwave/rail_zwave.h`

timings

RAIL_StateTiming_t RAIL_ZWAVE_Config_t::timings

Defines state timings for Z-Wave.

Definition at line 273 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_LrAckData_t

Configuration structure for Z-Wave Long Range Ack.

Public Attributes

- int8_t

[noiseFloorDbm](#)

Radio noise level measured on the channel the frame is transmitted on.
- int8_t

[txPowerDbm](#)

Transmit power used to transmit the ongoing Z-Wave Long Range Ack.
- int8_t

[receiveRssiDbm](#)

Signal strength measured while receiving the Z-Wave Long Range frame.

Public Attribute Documentation

noiseFloorDbm

```
int8_t RAIL_ZWAVE_LrAckData_t::noiseFloorDbm
```

Radio noise level measured on the channel the frame is transmitted on.

Definition at line 443 of file `protocol/zwave/rail_zwave.h`

txPowerDbm

```
int8_t RAIL_ZWAVE_LrAckData_t::txPowerDbm
```

Transmit power used to transmit the ongoing Z-Wave Long Range Ack.

Definition at line 445 of file `protocol/zwave/rail_zwave.h`

receiveRssiDbm

```
int8_t RAIL_ZWAVE_LrAckData_t::receiveRssiDbm
```

Signal strength measured while receiving the Z-Wave Long Range frame.

Definition at line 447 of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_BeamRxConfig_t

Configuration structure for Z-Wave beam detection.

Warnings

- This structure should not be used without direct instruction by Silicon Labs. Appropriate defaults for this are built into the RAIL library.

Public Attributes

RAIL_RxChannelHoppingConfig_t	channelHoppingConfig Channel hopping pattern to use for beam detection.
RAIL_RxDutyCycleConfig_t	receiveConfig_100 Amount of time to spend trying to receive a beam once detected.
RAIL_RxDutyCycleConfig_t	receiveConfig_40 Amount of time to spend trying to receive a beam once detected.

Public Attribute Documentation

channelHoppingConfig

RAIL_RxChannelHoppingConfig_t RAIL_ZWAVE_BeamRxConfig_t::channelHoppingConfig

Channel hopping pattern to use for beam detection.

Definition at line 460 of file protocol/zwave/rail_zwave.h

receiveConfig_100

RAIL_RxDutyCycleConfig_t RAIL_ZWAVE_BeamRxConfig_t::receiveConfig_100

Amount of time to spend trying to receive a beam once detected.

100kbps only

Definition at line 463 of file protocol/zwave/rail_zwave.h

receiveConfig_40

RAIL_RxDutyCycleConfig_t RAIL_ZWAVE_BeamRxConfig_t::receiveConfig_40

Amount of time to spend trying to receive a beam once detected.

40kbps only

Definition at line 466 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_RegionConfig_t

Each Z-Wave region supports 3 channels.

Public Attributes

<code>uint32_t</code>	<code>frequency</code> Channel frequency in hertz.
<code>RAIL_TxPower_t</code>	<code>maxPower</code> The maximum power allowed on the channel, in dBm.
<code>RAIL_ZWAVE_Baud_t</code>	<code>baudRate</code> Channel baud rate index.
<code>RAIL_ZWAVE_RegionId_t</code>	<code>regionId</code> Identification number for the region.
<code>RAIL_ZWAVE_RegionOptions_t</code>	<code>regionSpecific</code> Encapsulates region-specific options.

Public Attribute Documentation

frequency

```
uint32_t RAIL_ZWAVE_RegionConfig_t::frequency[(4U)]
```

Channel frequency in hertz.

Definition at line 480 of file `protocol/zwave/rail_zwave.h`

maxPower

```
RAIL_TxPower_t RAIL_ZWAVE_RegionConfig_t::maxPower[(4U)]
```

The maximum power allowed on the channel, in dBm.

Definition at line 482 of file `protocol/zwave/rail_zwave.h`

baudRate

```
RAIL_ZWAVE_Baud_t RAIL_ZWAVE_RegionConfig_t::baudRate[(4U)]
```

Channel baud rate index.

Definition at line 484 of file `protocol/zwave/rail_zwave.h`

regionId

RAIL_ZWAVE_RegionId_t RAIL_ZWAVE_RegionConfig_t::regionId

Identification number for the region.

Definition at line 486 of file protocol/zwave/rail_zwave.h

regionSpecific

RAIL_ZWAVE_RegionOptions_t RAIL_ZWAVE_RegionConfig_t::regionSpecific

Encapsulates region-specific options.

Definition at line 488 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_IrcalVal_t

Structure for Z-Wave Image Rejection Calibration.

Note

- Index 0 will hold the low side image rejection calibration value (channel 0), while index 1 will hold the high side image rejection value (channel 1).

Public Attributes

[RAIL_IrCalValues_t](#) [imageRejection](#)
Low side and high side image rejection values.

Public Attribute Documentation

imageRejection

RAIL_IrCalValues_t RAIL_ZWAVE_IrcalVal_t::imageRejection[2]

Low side and high side image rejection values.

Definition at line 500 of file protocol/zwave/rail_zwave.h

RF Sense

RF Sense

Modules

[RAIL_RfSenseSelectiveOokConfig_t](#)

Enumerations

```
enum RAIL\_RfSenseBand\_t {  
    RAIL_RFSENSE_OFF  
    RAIL_RFSENSE_2_4GHZ  
    RAIL_RFSENSE_SUBGHZ  
    RAIL_RFSENSE_ANY  
    RAIL_RFSENSE_MAX  
    RAIL_RFSENSE_2_4GHZ_LOW_SENSITIVITY = (0×20U) + RAIL_RFSENSE_2_4GHZ  
    RAIL_RFSENSE_SUBGHZ_LOW_SENSITIVITY = (0×20U) + RAIL_RFSENSE_SUBGHZ  
    RAIL_RFSENSE_ANY_LOW_SENSITIVITY = (0×20U) + RAIL_RFSENSE_ANY  
}  
An enumeration for specifying the RF Sense frequency band.
```

Typedefs

```
typedef void(* RAIL\_RfSense\_CallbackPtr\_t)(void)  
A pointer to an RF Sense callback function.
```

Functions

RAIL_Time_t	RAIL_StartRfSense (RAIL_Handle_t railHandle, RAIL_RfSenseBand_t band, RAIL_Time_t senseTime, RAIL_RfSense_CallbackPtr_t cb) Start/stop the RF Sense functionality in Energy Detection Mode for use during low-energy sleep modes.
RAIL_Status_t	RAIL_StartSelectiveOokRfSense (RAIL_Handle_t railHandle, RAIL_RfSenseSelectiveOokConfig_t *config) Start/stop the RF Sense functionality in Selective(OOK Based) Mode for use during low-energy sleep modes.
RAIL_Status_t	RAIL_ConfigRfSenseSelectiveOokWakeupPhy (RAIL_Handle_t railHandle) Switch to RF Sense Selective(OOK) PHY.
RAIL_Status_t	RAIL_SetRfSenseSelectiveOokWakeupPayload (RAIL_Handle_t railHandle, uint8_t numSyncwordBytes, uint32_t syncword) Set the transmit payload for waking up a node configured for RF Sense Selective(OOK).
bool	RAIL_IsRfSensed (RAIL_Handle_t railHandle) Check whether the RF was sensed.

Macros

```
#define RAIL_RFSENSE_LOW_SENSITIVITY_OFFSET (0×20U)  
RF Sense low sensitivity offset.
```

```
#define RAIL_RFSENSE_USE_HW_SYNCWORD (0U)
// Use the MODEM default sync word.
```

Enumeration Documentation

RAIL_RfSenseBand_t

```
RAIL_RfSenseBand_t
```

An enumeration for specifying the RF Sense frequency band.

Enumerator	
RAIL_RFSENSE_OFF	RF Sense is disabled.
RAIL_RFSENSE_2_4GHZ	RF Sense is in 2.4 GHz band.
RAIL_RFSENSE_SUBGHZ	RF Sense is in Sub-GHz band.
RAIL_RFSENSE_ANY	RF Sense is in both bands.
RAIL_RFSENSE_MAX	A count of the basic choices in this enumeration.
RAIL_RFSENSE_2_4GHZ_LOW_SENSITIVITY	RF Sense is in low sensitivity 2.4 GHz band.
RAIL_RFSENSE_SUBGHZ_LOW_SENSITIVITY	RF Sense is in low sensitivity Sub-GHz band.
RAIL_RFSENSE_ANY_LOW_SENSITIVITY	RF Sense is in low sensitivity for both bands.

Definition at line 4851 of file common/rail_types.h

Typedef Documentation

RAIL_RfSense_CallbackPtr_t

```
typedef void(* RAIL_RfSense_CallbackPtr_t) (void) (void)
```

A pointer to an RF Sense callback function.

Consider using the event [RAIL_EVENT_RF_SENSED](#) as an alternative.

Definition at line 4840 of file common/rail_types.h

Function Documentation

RAIL_StartRfSense

```
RAIL_Time_t RAIL_StartRfSense (RAIL_Handle_t railHandle, RAIL_RfSenseBand_t band, RAIL_Time_t senseTime,
RAIL_RfSense_CallbackPtr_t cb)
```

Start/stop the RF Sense functionality in Energy Detection Mode for use during low-energy sleep modes.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	band	The frequency band(s) on which to sense the RF energy. To stop RF Sense, specify RAIL_RFSENSE_OFF .
[in]	senseTime	The time (in microseconds) the RF energy must be continually detected to be considered "sensed".

[in]	cb	<code>RAIL_RfSense_CallbackPtr_t</code> is called when the RF is sensed. Set NULL if using <code>RAIL_EVENT_RF_SENSED</code> or polling via <code>RAIL_IsRfSensed()</code> .

Returns

- The actual `senseTime` used, which may be different than requested due to limitations of the hardware. If 0, RF sense was disabled or could not be enabled (no callback or event will be issued).

Some EFR32 radios have the ability to sense the presence of RF energy above -20 dBm within either or both the 2.4 GHz and Sub-GHz bands and trigger an event if that energy is continuously present for certain durations of time. An application can check when RF energy is sensed either by enabling the event `RAIL_EVENT_RF_SENSED`, by polling on the `RAIL_IsRfSensed()` API, or by using the cb callback.

Note

- After RF energy has been sensed, the RF Sense is automatically disabled and `RAIL_StartRfSense()` must be called again to reactivate it. If RF energy has not been sensed and to manually disable RF Sense, `RAIL_StartRfSense()` must be called with band specified as `RAIL_RFSENSE_OFF` or with `senseTime` set to 0 microseconds.
- Packet reception is not guaranteed to work correctly once RF Sense is enabled, both in single protocol and multiprotocol RAIL. To be safe, an application should turn this on only after idling the radio to stop receive and turn it off before attempting to restart receive. Since EM4 sleep causes the chip to come up through the reset vector any wake from EM4 must also shut off RF Sense to ensure proper receive functionality.

Warnings

- For some radios, RF Sense functionality is only guaranteed within a specified temperature range. See chip-specific documentation for more details.

Definition at line 5476 of file `common/rail.h`

RAIL_StartSelectiveOokRfSense

```
RAIL_Status_t RAIL_StartSelectiveOokRfSense (RAIL_Handle_t railHandle, RAIL_RfSenseSelectiveOokConfig_t * config)
```

Start/stop the RF Sense functionality in Selective(OOK Based) Mode for use during low-energy sleep modes.

Parameters

[in]	<code>railHandle</code>	A RAIL instance handle.
[in]	<code>config</code>	A pointer to a <code>RAIL_RfSenseSelectiveOokConfig_t</code> which holds the RF Sense configuration for Selective(OOK) mode.

Returns

- Status code indicating success of the function call.

Some radios support Selective RF energy detection (OOK mode) where the user can program the radio to look for a particular sync word pattern (1-4 bytes) sent using OOK and wake only when that is detected. See chip-specific documentation for more details.

The following code gives an example of how to use RF Sense functionality in Selective(OOK Based) Mode.

```
// Sync word length in bytes, 1-4 bytes.
#define NUM_SYNC_WORD_BYTES (2U)
// Sync word value.
#define SYNC_WORD            (0xB16FU)
// Desired RF band
RAIL_RfSenseBand_t rfBand = RAIL_RFSENSE_2_4GHZ;

// Configure the transmitting node for sending the wakeup packet.
RAIL_Idle(railHandle, RAIL_IDLE_ABORT, true);
RAIL_ConfigRfSenseSelectiveOokWakeupPhy(railHandle);
```



```
RAIL_SetRfSenseSelectiveOokWakeupPayload(railHandle, NUM_SYNC_WORD_BYTES, SYNC_WORD);RAIL_StartTx(railHandle, channel,
RAIL_TX_OPTIONS_DEFAULT, NULL);// Configure the receiving node (EFR32xG22) for RF Sense.
RAIL_RfSenseSelectiveOokConfig_t config = {.band = rfBand,.syncWordNumBytes = NUM_SYNC_WORD_BYTES,.syncWord = SYNC_WORD,.cb =
NULL // Use RAIL_EVENT_RF_SENSED event or poll RAIL_IsRfSensed()};RAIL_StartSelectiveOokRfSense(railHandle,&config);
```

Note

- After RF energy has been sensed, the RF Sense is automatically disabled and [RAIL_StartSelectiveOokRfSense\(\)](#) must be called again to reactivate. If RF energy has not been sensed and to manually disable RF Sense, [RAIL_StartSelectiveOokRfSense\(\)](#) must be called with band specified as [RAIL_RFSENSE_OFF](#) or with [RAIL_RfSenseSelectiveOokConfig_t](#) as NULL.
- Packet reception is not guaranteed to work correctly once RF Sense is enabled, both in single protocol and multiprotocol RAIL. To be safe, an application should turn this on only after idling the radio to stop receive and turn it off before attempting to restart receive. Since EM4 sleep causes the chip to come up through the reset vector any wake from EM4 must also shut off RF Sense to ensure proper receive functionality.

Definition at line 5535 of file common/rail.h

RAIL_ConfigRfSenseSelectiveOokWakeupPhy

```
RAIL_Status_t RAIL_ConfigRfSenseSelectiveOokWakeupPhy (RAIL_Handle_t railHandle)
```

Switch to RF Sense Selective(OOK) PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- Status code indicating success of the function call.

This function switches to the RF Sense Selective(OOK) PHY for transmitting a packet to wake up a chip that supports Selective RF energy detection (OOK mode). You may only call this function while the radio is idle. While the radio is configured for this PHY, receive functionality should not be used.

Note

- The user must also set up the transmit FIFO, via [RAIL_SetRfSenseSelectiveOokWakeupPayload](#), post this function call to include the first byte as the Preamble byte, followed by the Sync word (1-4 bytes). See chip-specific documentation for more details.

Definition at line 5555 of file common/rail.h

RAIL_SetRfSenseSelectiveOokWakeupPayload

```
RAIL_Status_t RAIL_SetRfSenseSelectiveOokWakeupPayload (RAIL_Handle_t railHandle, uint8_t numSyncwordBytes,
uint32_t syncword)
```

Set the transmit payload for waking up a node configured for RF Sense Selective(OOK).

Parameters

[in]	railHandle	A handle for RAIL instance.
[in]	numSyncwordBytes	Sync word length in bytes, 1-4 bytes.
[in]	syncword	Sync word value.

Returns

- Status code indicating success of the function call.

Note

- You must call this function after the chip has been set up with the RF Sense Selective(OOK) PHY, using [RAIL_ConfigRfSenseSelectiveOokWakeupPhy](#).

Definition at line 5569 of file common/rail.h

RAIL_IsRfSensed

```
bool RAIL_IsRfSensed (RAIL_Handle_t railHandle)
```

Check whether the RF was sensed.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if RF was sensed since the last call to [RAIL_StartRfSense\(\)](#). false otherwise.

This function is useful if [RAIL_StartRfSense\(\)](#) is called with a NULL callback. It is generally used after EM4 reboot but can be used any time.

Definition at line 5583 of file common/rail.h

RAIL_RfSenseSelectiveOokConfig_t

Structure to configure RFSENSE Selective(OOK) mode.

Public Attributes

RAIL_RfSenseBand_t	band
	The frequency band(s) on which to sense the RF energy.
uint8_t	syncWordNumBytes
	Sync word length in bytes, 1-4 bytes.
uint32_t	syncWord
	Sync Word Value.
RAIL_RfSense_CallbackPtr_t	cb
	The callback function, called when RF is sensed.

Public Attribute Documentation

band

RAIL_RfSenseBand_t RAIL_RfSenseSelectiveOokConfig_t::band

The frequency band(s) on which to sense the RF energy.

To stop RF Sense, specify [RAIL_RFSENSE_OFF](#).

Definition at line 4901 of file common/rail_types.h

syncWordNumBytes

uint8_t RAIL_RfSenseSelectiveOokConfig_t::syncWordNumBytes

Sync word length in bytes, 1-4 bytes.

Note

- When [syncWord](#) is set to use [RAIL_RFSENSE_USE_HW_SYNCWORD](#), the [syncWordNumBytes](#) value will be ignored since we rely on the HW default settings for sync word.

Definition at line 4908 of file common/rail_types.h

syncWord

uint32_t RAIL_RfSenseSelectiveOokConfig_t::syncWord

Sync Word Value.

To use HW default sync word, set to [RAIL_RFSENSE_USE_HW_SYNCWORD](#).

Definition at line 4913 of file common/rail_types.h

cb

```
RAIL_RfSense_CallbackPtr_t RAIL_RfSenseSelectiveOokConfig_t::cb
```

The callback function, called when RF is sensed.

Note

- Set to NULL and instead use [RAIL_EVENT_RF_SENSED](#) or poll via [RAIL_IsRfSensed\(\)](#).

Definition at line 4920 of file `common/rail_types.h`

RX Channel Hopping

RX Channel Hopping

Hardware accelerated hopping between channels while waiting for a packet in receive.

Channel hopping provides a hardware accelerated method for scanning across multiple channels quickly, as part of a receive protocol. While it is possible to call [RAIL_StartRx\(\)](#) on different channels, back to back, and listen on many channels sequentially in that way, the time it takes to switch channels with that method may be too long for some protocols. This API pre-computes necessary channel change operations for a given list of channels, so that the radio can move from channel to channel much faster. Additionally, it leads to more succinct code as channel changes will be done implicitly, without requiring numerous calls to [RAIL_StartRx\(\)](#). Currently, while this feature is enabled, the radio will hop channels in the given sequence each time it enters RX. Note that RX Channel hopping and EFR32xG25's concurrent mode / collision detection are mutually exclusive.

The channel hopping buffer requires [RAIL_CHANNEL_HOPPING_BUFFER_SIZE_PER_CHANNEL](#) number of 32-bit words of overhead per channel, plus 3 words overall plus the twice the size of the [RAIL_ChannelConfig_t::phyConfigDeltaSubtract](#) of the whole radio configuration, plus the twice the sum of the sizes of all the [RAIL_ChannelConfigEntry_t::phyConfigDeltaAdd](#) in all the channel hopping channels.

The following code gives an example of how to use the RX Channel Hopping API.

```
#define CHANNEL_HOPPING_NUMBER_OF_CHANNELS 4
#define CHANNEL_HOPPING_BUFFER_SIZE do { \
    3 + \
    (RAIL_CHANNEL_HOPPING_BUFFER_SIZE_PER_CHANNEL \
     * CHANNEL_HOPPING_NUMBER_OF_CHANNELS) + \
    2 * (sizeof(UINT32_DELTA_SUBTRACT) + \
        sizeof(UINT32_DELTA_ADD_0) + \
        sizeof(UINT32_DELTA_ADD_1) + \
        sizeof(UINT32_DELTA_ADD_2) + \
        sizeof(UINT32_DELTA_ADD_3)) \
} while (0)

RAIL_RxChannelHoppingConfigEntry_t channelHoppingEntries[CHANNEL_HOPPING_NUMBER_OF_CHANNELS];
uint32_t channelHoppingBuffer[CHANNEL_HOPPING_BUFFER_SIZE];

RAIL_RxChannelHoppingConfig_t channelHoppingConfig = {
    .buffer = channelHoppingBuffer,
    .bufferLength = CHANNEL_HOPPING_BUFFER_SIZE,
    .numberOfChannels = CHANNEL_HOPPING_NUMBER_OF_CHANNELS,
    .entries = channelHoppingEntries
};

channelHoppingEntries[0].channel = 1;
channelHoppingEntries[1].channel = 2;
channelHoppingEntries[2].channel = 3;

RAIL_ConfigRxChannelHopping(railHandle, &channelHoppingConfig);
RAIL_EnableRxChannelHopping(railHandle, true, true)
```

Modules

[RAIL_RxChannelHoppingConfigMultiMode_t](#)

[RAIL_RxChannelHoppingConfigEntry_t](#)

[RAIL_RxChannelHoppingConfig_t](#)
[RAIL_RxDutyCycleConfig_t](#)
[EFR32xG2x](#)
[S1xx3xx](#)

Enumerations

```
enum RAIL\_RxChannelHoppingMode\_t {
    RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL = 0
    RAIL_RX_CHANNEL_HOPPING_MODE_TIMEOUT = 1
    RAIL_RX_CHANNEL_HOPPING_MODE_TIMING_SENSE = 2
    RAIL_RX_CHANNEL_HOPPING_MODE_PREAMBLE_SENSE = 3
    RAIL_RX_CHANNEL_HOPPING_MODE_RESERVED1 = 4
    RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE = 5
    RAIL_RX_CHANNEL_HOPPING_MODE_SQ = 6
    RAIL_RX_CHANNEL_HOPPING_MODE_CONC = 7
    RAIL_RX_CHANNEL_HOPPING_MODE_VT = 8
    RAIL_RX_CHANNEL_HOPPING_MODE_TX = 9
    RAIL_RX_CHANNEL_HOPPING_MODES_COUNT
    RAIL_RX_CHANNEL_HOPPING_MODES_WITH_OPTIONS_BASE = 0x80
    RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL_WITH_OPTIONS =
        (RAIL_RX_CHANNEL_HOPPING_MODES_WITH_OPTIONS_BASE +
        RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL)
    RAIL_RX_CHANNEL_HOPPING_MODE_TIMEOUT_WITH_OPTIONS =
        (RAIL_RX_CHANNEL_HOPPING_MODES_WITH_OPTIONS_BASE +
        RAIL_RX_CHANNEL_HOPPING_MODE_TIMEOUT)
    RAIL_RX_CHANNEL_HOPPING_MODE_TIMING_SENSE_WITH_OPTIONS =
        (RAIL_RX_CHANNEL_HOPPING_MODES_WITH_OPTIONS_BASE +
        RAIL_RX_CHANNEL_HOPPING_MODE_TIMING_SENSE)
    RAIL_RX_CHANNEL_HOPPING_MODE_PREAMBLE_SENSE_WITH_OPTIONS =
        (RAIL_RX_CHANNEL_HOPPING_MODES_WITH_OPTIONS_BASE +
        RAIL_RX_CHANNEL_HOPPING_MODE_PREAMBLE_SENSE)
    RAIL_RX_CHANNEL_HOPPING_MODE_RESERVED1_WITH_OPTIONS =
        (RAIL_RX_CHANNEL_HOPPING_MODES_WITH_OPTIONS_BASE +
        RAIL_RX_CHANNEL_HOPPING_MODE_RESERVED1)
    RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE_WITH_OPTIONS =
        (RAIL_RX_CHANNEL_HOPPING_MODES_WITH_OPTIONS_BASE +
        RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE)
    RAIL_RX_CHANNEL_HOPPING_MODE_SQ_WITH_OPTIONS =
        (RAIL_RX_CHANNEL_HOPPING_MODES_WITH_OPTIONS_BASE + RAIL_RX_CHANNEL_HOPPING_MODE_SQ)
    RAIL_RX_CHANNEL_HOPPING_MODE_CONC_WITH_OPTIONS =
        (RAIL_RX_CHANNEL_HOPPING_MODES_WITH_OPTIONS_BASE +
        RAIL_RX_CHANNEL_HOPPING_MODE_CONC)
    RAIL_RX_CHANNEL_HOPPING_MODE_VT_WITH_OPTIONS =
        (RAIL_RX_CHANNEL_HOPPING_MODES_WITH_OPTIONS_BASE + RAIL_RX_CHANNEL_HOPPING_MODE_VT)
    RAIL_RX_CHANNEL_HOPPING_MODE_TX_WITH_OPTIONS =
        (RAIL_RX_CHANNEL_HOPPING_MODES_WITH_OPTIONS_BASE + RAIL_RX_CHANNEL_HOPPING_MODE_TX)
}
Modes by which RAIL can determine when to proceed to the next channel during channel hopping.
```

```
enum RAIL\_RxChannelHoppingDelayMode\_t {
    RAIL_RX_CHANNEL_HOPPING_DELAY_MODE_STATIC = 0
}
```

```
enum RAIL\_RxChannelHoppingOptions\_t {
    RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_SYNTH_CAL_SHIFT = 0
    RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_DC_CAL_SHIFT = 1
    RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD_SHIFT = 2
    RAIL_RX_CHANNEL_HOPPING_OPTION_STOP_SHIFT = 3
    RAIL_RX_CHANNEL_HOPPING_OPTIONS_COUNT
}
Options that can customize channel hopping behavior on a per-hop basis.
```

Typedefs

typedef uint32_t [RAIL_RxChannelHoppingParameter_t](#)
Rx channel hopping on-channel time.

Functions

[RAIL_Status_t](#) [RAIL_ConfigRxChannelHopping](#)(RAIL_Handle_t railHandle, RAIL_RxChannelHoppingConfig_t *config)
Configure RX channel hopping.

[RAIL_Status_t](#) [RAIL_EnableRxChannelHopping](#)(RAIL_Handle_t railHandle, bool enable, bool reset)
Enable RX channel hopping.

[RAIL_Status_t](#) [RAIL_TriggerRxChannelHop](#)(RAIL_Handle_t railHandle)
Manually trigger an Rx channel hop.

int16_t [RAIL_GetChannelHoppingRssi](#)(RAIL_Handle_t railHandle, uint8_t channelIndex)
Get RSSI in quarter-dBm of one channel in the channel hopping sequence, during channel hopping.

[RAIL_Status_t](#) [RAIL_ConfigRxDutyCycle](#)(RAIL_Handle_t railHandle, const RAIL_RxDutyCycleConfig_t *config)
Configure RX duty cycle mode.

[RAIL_Status_t](#) [RAIL_EnableRxDutyCycle](#)(RAIL_Handle_t railHandle, bool enable)
Enable RX duty cycle mode.

[RAIL_Status_t](#) [RAIL_GetDefaultRxDutyCycleConfig](#)(RAIL_Handle_t railHandle, RAIL_RxDutyCycleConfig_t *config)
Get the default RX duty cycle configuration.

Macros

#define RAIL_RX_CHANNEL_HOPPING_MAX_SENSE_TIME_US 0x08000000UL
The maximum sense time supported for those [RAIL_RxChannelHoppingMode_t](#) modes whose parameter(s) specify a sensing time.

#define RAIL_RX_CHANNEL_HOPPING_OPTIONS_NONE 0U
A value representing no options enabled.

#define RAIL_RX_CHANNEL_HOPPING_OPTIONS_DEFAULT RAIL_RX_CHANNEL_HOPPING_OPTIONS_NONE
All options disabled by default.

#define RAIL_RX_CHANNEL_HOPPING_OPTION_DEFAULT RAIL_RX_CHANNEL_HOPPING_OPTIONS_DEFAULT

#define RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_SYNTH_CAL (1U << RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_SYNTH_CAL_SHIFT)
An option to skip synth calibration while **hopping into** the channel specified in the current entry.

#define RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_DC_CAL (1U << RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_DC_CAL_SHIFT)
An option to skip DC calibration while **hopping into** the channel specified in the current entry.

#define RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD (1U << RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD_SHIFT)
An option to check RSSI after **hopping into** the channel specified in the current entry and hop if that RSSI is below the threshold specified in [RAIL_RxChannelHoppingConfigEntry_t::rssiThresholdDbm](#).

#define RAIL_RX_CHANNEL_HOPPING_OPTION_STOP (1U << RAIL_RX_CHANNEL_HOPPING_OPTION_STOP_SHIFT)
An option to stop the hopping sequence at this entry in the hop table, which also idles the radio.

#define RAIL_CHANNEL_HOPPING_INVALID_INDEX (0xFEU)
A sentinel value to flag an invalid channel hopping index.

Enumeration Documentation

RAIL_RxChannelHoppingMode_t

RAIL_RxChannelHoppingMode_t

Modes by which RAIL can determine when to proceed to the next channel during channel hopping.

Enumerator	
RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL	Switch to the next channel each time the radio re-enters RX after packet reception or a transmit based on the corresponding State Transitions .
RAIL_RX_CHANNEL_HOPPING_MODE_TIMEOUT	Switch to the next channel after a certain amount of time passes.
RAIL_RX_CHANNEL_HOPPING_MODE_TIMING_SENSE	Listen in receive RX for at least a specified timeout.
RAIL_RX_CHANNEL_HOPPING_MODE_PREAMBLE_SENSE	Listen in receive RX for at least a specified timeout.
RAIL_RX_CHANNEL_HOPPING_MODE_RESERVED1	Placeholder for a reserved hopping mode that is not supported.
RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE	A mode that combines modes TIMING_SENSE, PREAMBLE_SENSE, and TIMEOUT (sync detect) all running in parallel.
RAIL_RX_CHANNEL_HOPPING_MODE_SQ	Switch to the next channel based on the demodulation settings in the PHY config.
RAIL_RX_CHANNEL_HOPPING_MODE_CONC	Marks that the channel is concurrent with another channel, and otherwise behaves identically to RAIL_RX_CHANNEL_HOPPING_MODE_SQ .
RAIL_RX_CHANNEL_HOPPING_MODE_VT	Indicates that this channel is a virtual channel that is concurrently detected with the channel indicated by the RAIL_RxChannelHoppingConfigEntry_t::parameter .
RAIL_RX_CHANNEL_HOPPING_MODE_TX	This is the transmit channel used for Auto-Ack if the regular channel, specified in RAIL_RxChannelHoppingConfigEntry::parameter , is optimized for RX which may degrade some TX performance.
RAIL_RX_CHANNEL_HOPPING_MODES_COUNT	A count of the basic choices in this enumeration.
RAIL_RX_CHANNEL_HOPPING_MODES_WITH_OPTIONS_BASE	The start of equivalent modes requiring non-default RAIL_RxDutyCycleConfig_t::options , needed for backwards-compatibility with earlier RAIL_RxDutyCycleConfig_t format.
RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL with options.
RAIL_RX_CHANNEL_HOPPING_MODE_TIMEOUT_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_TIMEOUT with options.
RAIL_RX_CHANNEL_HOPPING_MODE_TIMING_SENSE_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_TIMING_SENSE with options.
RAIL_RX_CHANNEL_HOPPING_MODE_PREAMBLE_SENSE_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_PREAMBLE_SENSE with options.

RAIL_RX_CHANNEL_HOPPING_MODE_RESERVED1_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_RESERVED1 with options.
RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE with options.
RAIL_RX_CHANNEL_HOPPING_MODE_SQ_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_SQ with options.
RAIL_RX_CHANNEL_HOPPING_MODE_CONC_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_CONC with options.
RAIL_RX_CHANNEL_HOPPING_MODE_VT_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_VT with options.
RAIL_RX_CHANNEL_HOPPING_MODE_TX_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_TX with options.

Definition at line 4938 of file common/rail_types.h

RAIL_RxChannelHoppingDelayMode_t

RAIL_RxChannelHoppingDelayMode_t

DeprecatedSet only to RAIL_RX_CHANNEL_DELAY_MODE_STATIC.

Enumerator

RAIL_RX_CHANNEL_HOPPING_DELAY_MODE_STATIC	Always delay for exactly the amount of time specified in the delay parameter, regardless of how other channel hopping channels were extended via preamble sense or other means.

Definition at line 5115 of file common/rail_types.h

RAIL_RxChannelHoppingOptions_t

RAIL_RxChannelHoppingOptions_t

Options that can customize channel hopping behavior on a per-hop basis.

Enumerator

RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_SYNTH_CAL_SHIFT	Shift position of RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_SYNTH_CAL bit.
RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_DC_CAL_SHIFT	Shift position of RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_DC_CAL bit.
RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD_SHIFT	Shift position of RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD bit.
RAIL_RX_CHANNEL_HOPPING_OPTION_STOP_SHIFT	Shift position of RAIL_RX_CHANNEL_HOPPING_OPTION_STOP bit.
RAIL_RX_CHANNEL_HOPPING_OPTIONS_COUNT	A count of the choices in this enumeration.

Definition at line 5140 of file common/rail_types.h

Typedef Documentation

RAIL_RxChannelHoppingParameter_t

RAIL_RxChannelHoppingParameter_t

Rx channel hopping on-channel time.

Definition at line 5133 of file common/rail_types.h

Function Documentation

RAIL_ConfigRxChannelHopping

RAIL_Status_t RAIL_ConfigRxChannelHopping (RAIL_Handle_t railHandle, RAIL_RxChannelHoppingConfig_t * config)

Configure RX channel hopping.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	A pointer to the configuration parameters for RX Channel Hopping.

Returns

- Status code indicating success of the function call.

Configure channel hopping channels, conditions, and parameters. This API must be called before [RAIL_EnableRxChannelHopping\(\)](#). This API must never be called while the radio is on with RX Duty Cycle or Channel Hopping enabled.

Note

- Use the compile time symbol [RAIL_SUPPORTS_CHANNEL_HOPPING](#) or the runtime call [RAIL_SupportsChannelHopping\(\)](#) to check whether the platform supports this feature.
- Calling this function will overwrite any settings configured with [RAIL_ConfigRxDutyCycle\(\)](#).

Definition at line 5670 of file common/rail.h

RAIL_EnableRxChannelHopping

RAIL_Status_t RAIL_EnableRxChannelHopping (RAIL_Handle_t railHandle, bool enable, bool reset)

Enable RX channel hopping.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	Enable (true) or disable (false) RX Channel Hopping.
[in]	reset	Start from the first channel of the channel hopping sequence (true) or from wherever hopping left off last time the code left RX.

Returns

- Status code indicating success of the function call.

Enable or disable Channel Hopping. Additionally, specify whether hopping should be reset to start from the first channel index, or continue from the channel last hopped to. The radio should not be on when this API is called. [RAIL_ConfigRxChannelHopping\(\)](#) must be called successfully before this API is called.

Note

- Use the compile time symbol [RAIL_SUPPORTS_CHANNEL_HOPPING](#) or the runtime call [RAIL_SupportsChannelHopping\(\)](#) to check whether the platform supports this feature.
- RX Channel Hopping may not be enabled while Auto-Acking is enabled.
- Calling this function will overwrite any settings configured with [RAIL_EnableRxDutyCycle\(\)](#).

Definition at line 5698 of file common/rail.h

RAIL_TriggerRxChannelHop

RAIL_Status_t RAIL_TriggerRxChannelHop ([RAIL_Handle_t](#) railHandle)

Manually trigger an Rx channel hop.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

This function facilitates a manual hop when using [RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL](#). (It will also trigger a hop in other automatic hop modes as well, so use with caution.)

Note

- This function may cause [RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE](#) to occur before it returns.

Definition at line 5715 of file common/rail.h

RAIL_GetChannelHoppingRssi

int16_t RAIL_GetChannelHoppingRssi ([RAIL_Handle_t](#) railHandle, uint8_t channelIndex)

Get RSSI in quarter-dBm of one channel in the channel hopping sequence, during channel hopping.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channelIndex	Index in the channel hopping sequence of the channel of interest.

Returns

- Latest RSSI in quarter-dBm (dBm * 4) for the channel at the specified index.

Note

- Use the compile time symbol [RAIL_SUPPORTS_CHANNEL_HOPPING](#) or the runtime call [RAIL_SupportsChannelHopping\(\)](#) to check whether the platform supports this feature.
- In multiprotocol, this function returns [RAIL_RSSI_INVALID](#) immediately if railHandle is not the currently active [RAIL_Handle_t](#).
- [RAIL_ConfigRxChannelHopping\(\)](#) must be called successfully before this API is called.
- When the Z-Wave protocol is active, running [RAIL_GetChannelHoppingRssi\(\)](#) on the 9.6 kbps PHY returns the RSSI measurement of the 40kbps PHY. This is because the 9.6 kbps PHY has trouble with RSSI measurements on EFR32xG2x

family of chips.

Definition at line 5742 of file common/rail.h

RAIL_ConfigRxDutyCycle

RAIL_Status_t RAIL_ConfigRxDutyCycle (RAIL_Handle_t railHandle, const RAIL_RxDutyCycleConfig_t * config)

Configure RX duty cycle mode.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	A non-NULL pointer to a configuration structure specifying duty cycle parameters.

Returns

- Status code indicating success of the function call.

Configure RX duty cycle mode. With this mode enabled, every time the radio enters RX, it will duty cycle on and off to save power. The duty cycle ratio can be altered dynamically and intelligently by the hardware by staying on longer if a preamble or other packet segments are detected in the air. This API must never be called while the radio is on with RX Duty Cycle or Channel Hopping enabled. For short delays (in the order of microseconds), `RAIL_RxDutyCycleConfig_t::delay`, this can be used to save receive current while having little impact on the radio performance, for protocols with long preambles. For long delays (in the order of milliseconds or higher) the chip can be put into EM2 energy mode before re-entering RX, to save extra power, with some application hooks as shown below.

```
#include "rail.h"

extern RAIL_Handle_t railHandle;
RAIL_Time_t periodicWakeupUs;

volatile bool dutyCycleEnd = true;

void RAILCb_Event(RAIL_Handle_t railHandle, RAIL_Events_t events)
{
    // Omitting other event handlers
    if (events & RAIL_EVENT_RX_DUTY_CYCLE_RX_END) {
        dutyCycleEnd = true;
    }
}

void main(void)
{
    RAIL_Status_t status;
    bool shouldSleep = false;

    initializeRail(); // Call RAIL_Init() and such

    // This function depends on your board/chip but it must enable the LFCLK
    // you intend to use for RTCC sync before we configure sleep as that
    // function will attempt to auto detect the clock.
    BoardSetupLFCLK();
    // Initialize Power Manager module
    sL_power_manager_init();
    // Initialize RAIL Power Manager
    RAIL_InitPowerManager();

    // Configure sleep for timer synchronization
    RAIL_TimerSyncConfig_t timerSyncConfig = RAIL_TIMER_SYNC_DEFAULT;
    status = RAIL_ConfigSleep(railHandle, &timerSyncConfig);
    assert(status == RAIL_STATUS_NO_ERROR);
```

```
// Configure RX duty-cycling
RAIL_RxDutyCycleConfig_t duty_cycle_config = {.mode = RAIL_RX_CHANNEL_HOPPING_MODE_PREAMBLE_SENSE, parameter =
DUTY_CYCLE_ON_TIME, delay_us = DUTY_CYCLE_OFF_TIME, delay_mode = RAIL_RX_CHANNEL_HOPPING_DELAY_MODE_STATIC, options =
RAIL_RX_CHANNEL_HOPPING_OPTIONS_NONE};
status = RAIL_ConfigRxDutyCycle(rail_handle, &duty_cycle_config); assert(status == RAIL_STATUS_NO_ERROR);
status = RAIL_EnableRxDutyCycle(rail_handle, true); assert(status == RAIL_STATUS_NO_ERROR); // Application main loop while(1){if(dutyCycleEnd){
dutyCycleEnd = false; // Schedule the next receive.
RAIL_ScheduleRxConfig_t rxConfig = {.start = periodicWakeupUs, startMode = RAIL_TIME_DELAY, end = 0U, endMode =
RAIL_TIME_DISABLED, rxTransitionEndSchedule = 0U, hardWindowEnd = 0U
}; RAIL_Idle(railHandle, RAIL_IDLE_ABORT, true); RAIL_StartScheduledRx(railHandle, channel, &rxConfig, NULL); // ... do normal app stuff and set
shouldSleep when we want to sleep if(shouldSleep){ // Let the CPU go to sleep if the system allows it. sl_power_manager_sleep(); }}
```

Note

- Use the compile time symbol `RAIL_SUPPORTS_CHANNEL_HOPPING` or the runtime call `RAIL_SupportsChannelHopping()` to check whether the platform supports this feature.
- Calling this function will overwrite any settings configured with `RAIL_ConfigRxChannelHopping()`.

Definition at line 5846 of file `common/rail.h`

RAIL_EnableRxDutyCycle

RAIL_Status_t RAIL_EnableRxDutyCycle (RAIL_Handle_t railHandle, bool enable)

Enable RX duty cycle mode.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	Enable (true) or disable (false) RX Duty Cycling.

Returns

- Status code indicating success of the function call.

Enable or disable RX duty cycle mode. After this is called, the radio will begin duty cycling each time it enters RX, based on the configuration passed to `RAIL_ConfigRxDutyCycle()`. This API must not be called while the radio is on.

Note

- Use the compile time symbol `RAIL_SUPPORTS_CHANNEL_HOPPING` or the runtime call `RAIL_SupportsChannelHopping()` to check whether the platform supports this feature.
- Calling this function will overwrite any settings configured with `RAIL_EnableRxChannelHopping()`.

Definition at line 5868 of file `common/rail.h`

RAIL_GetDefaultRxDutyCycleConfig

RAIL_Status_t RAIL_GetDefaultRxDutyCycleConfig (RAIL_Handle_t railHandle, RAIL_RxDutyCycleConfig_t * config)

Get the default RX duty cycle configuration.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	config	An application-provided non-NULL pointer to store the default RX duty cycle configuration.

Returns

Status code indicating success of the function call. Note that `RAIL_STATUS_INVALID_PARAMETER` will be returned if the current channel's radio configuration does not support the requested information.

To save power during RX, an application may want to go to low power as long as possible by periodically waking up and trying to "sense" if there are any incoming packets. This API returns the recommended RX duty cycle configuration, so the application can enter low power mode periodically without missing packets. To wake up earlier, the application can reduce the delay parameter. Note that these value might be different if any configuration / channel has changed.

Definition at line 5890 of file common/rail.h

RAIL_RxChannelHoppingConfigMultiMode_t

Structure that parameterizes [RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE](#).

Every [RAIL_RxChannelHoppingConfigEntry_t](#) or [RAIL_RxDutyCycleConfig_t](#) that uses [RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE](#) must allocate one of these structures in global read-write memory to provide the settings for this mode and for RAIL to use during hopping or duty cycling. A pointer to this structure, cast appropriately, is what is passed in the corresponding [RAIL_RxChannelHoppingConfigEntry_t::parameter](#) or [RAIL_RxDutyCycleConfig_t::parameter](#).

The contents of this structure must be initialized prior to each [RAIL_ConfigRxChannelHopping\(\)](#) or [RAIL_ConfigRxDutyCycle\(\)](#) call and must not be touched thereafter until the next such call. RAIL may change these contents during configuration or operation.

This mode of operation functions algorithmically like this pseudocode:

```

extern bool channelHopping; // true if channel hopping, false if duty cycling
extern RAIL_RxChannelHoppingConfigEntry_t *hopConfigEntry; // current channel

static RAIL_RxChannelHoppingConfigMultiMode_t *multiParams;
static RAIL_Time_t rxStartTime;
static bool preambleSensed;

static void hopOrSuspendRx(uint32_t delay)
{
    disableDemodEvents();
    disableTimerEvents();
    stopTimer();
    if (channelHopping) {
        hopToNextChannel(delay, &hopConfigEntry); // updates hopConfigEntry
    } else {
        suspendRx(delay);
    }
    onStartRx(); // resume receive after delay (on new channel if hopping)
}

void onStartRx(void) // called upon entry to receive
{
    rxStartTime = RAIL_GetTime();
    multiParams = (RAIL_RxChannelHoppingConfigMultiMode_t *)
        (void *)hopConfigEntry->parameter;
    startTimer(rxStartTime + multiParams->timingSense);
    preambleSensed = false;
    enableTimerEvents(); // timer will trigger onTimerEvent() handler
    enableDemodEvents(); // demod will trigger onDemodEvent() handler
}

void onTimerEvent(void) // called when timer expires
{
    hopOrSuspendRx(hopConfigEntry->delay);
}

void onDemodEvent(void) // called when demodulator state changes
{
    if (DEMOD_TIMING_SENSED) {
        stopTimer();
        startTimer(rxStartTime + multiParams->syncDetect);
    }
    if (DEMOD_TIMING_LOST) {
        stopTimer();
        uint32_t newTimeout = RAIL_GetTime() + multiParams->timingReSense;
        uint32_t limitTimeout;
        if (preambleSensed) {
            limitTimeout = rxStartTime + multiParams->syncDetect;
        } else {
            limitTimeout = rxStartTime + multiParams->preambleSense;
        }
        if (newTimeout > limitTimeout) {
            newTimeout = limitTimeout;
        }
        if (newTimeout > RAIL_GetTime()) {
            startTimer(newTimeout);
        } else {
            hopOrSuspendRx(hopConfigEntry->delay);
        }
    }
    if (DEMOD_PREAMBLE_SENSED) {
        preambleSensed = true;
    }
    if (DEMOD_PREAMBLE_LOST) {
        preambleSensed = false;
    }
}

```



```
disableDemodEvents();disableTimerEvents();stopTimer();receivePacket();// stay on channel to receive framehopOrSuspendRx(0);// continue RX
per state transitions with no delay}}
```

Public Attributes

uint32_t	syncDetect	Switch to the next channel if sync is not detected before this time, in microseconds, measured from entry to Rx.
uint32_t	preambleSense	Switch to the next channel if timing was sensed but then lost after this time, in microseconds, measured from entry to Rx – unless preamble had been sensed in which case any switching is deferred to timingReSense and, if timing is regained, to syncDetect.
uint32_t	timingSense	Switch to the next channel if timing is not sensed before this time, in microseconds, measured from entry to Rx.
uint32_t	timingReSense	Switch to the next channel if timing was sensed but then lost and not regained before this time, in microseconds, measured from when timing was lost.
uint32_t	status	Set this to 0.

Public Attribute Documentation

syncDetect

```
uint32_t RAIL_RxChannelHoppingConfigMultiMode_t::syncDetect
```

Switch to the next channel if sync is not detected before this time, in microseconds, measured from entry to Rx.

This must be greater than preambleSense and less than [RAIL_RX_CHANNEL_HOPPING_MAX_SENSE_TIME_US](#).

Definition at line 5293 of file `common/rail_types.h`

preambleSense

```
uint32_t RAIL_RxChannelHoppingConfigMultiMode_t::preambleSense
```

Switch to the next channel if timing was sensed but then lost after this time, in microseconds, measured from entry to Rx – unless preamble had been sensed in which case any switching is deferred to timingReSense and, if timing is regained, to syncDetect.

This must be greater than timingSense and less than [RAIL_RX_CHANNEL_HOPPING_MAX_SENSE_TIME_US](#).

Definition at line 5302 of file `common/rail_types.h`

timingSense

```
uint32_t RAIL_RxChannelHoppingConfigMultiMode_t::timingSense
```

Switch to the next channel if timing is not sensed before this time, in microseconds, measured from entry to Rx.

This must be greater than 2 and less than [RAIL_RX_CHANNEL_HOPPING_MAX_SENSE_TIME_US](#).

Definition at line 5309 of file `common/rail_types.h`

timingReSense

```
uint32_t RAIL_RxChannelHoppingConfigMultiMode_t::timingReSense
```

Switch to the next channel if timing was sensed but then lost and not regained before this time, in microseconds, measured from when timing was lost.

This must be less than [RAIL_RX_CHANNEL_HOPPING_MAX_SENSE_TIME_US](#).

Definition at line 5316 of file `common/rail_types.h`

status

```
uint32_t RAIL_RxChannelHoppingConfigMultiMode_t::status
```

Set this to 0.

This field, along with the others, may be used internally by RAIL during configuration or operation.

Definition at line 5321 of file `common/rail_types.h`

RAIL_RxChannelHoppingConfigEntry_t

Structure that represents one of the channels that is part of a [RAIL_RxChannelHoppingConfig_t](#) sequence of channels used in channel hopping.

Public Attributes

uint16_t	channel	The channel number to be used for this entry in the channel hopping sequence.
RAIL_RxChannelHoppingMode_t	mode	The mode by which RAIL determines when to hop to the next channel.
RAIL_RxChannelHoppingParameter_t	parameter	Depending on the 'mode' parameter that was specified, this member is used to parameterize that mode.
uint32_t	delay	Idle time in microseconds to wait before hopping into the channel indicated by this entry.
RAIL_RxChannelHoppingDelayMode_t	delayMode	
RAIL_RxChannelHoppingOptions_t	options	Bitmask of various options that can be applied to the current channel hop.
int8_t	rssiThresholdDbm	The RSSI threshold (in dBm) below which a hop will occur in any mode when RAIL_RX_CHANNEL_HOPPING_OPTION_RSSLTHRESHOLD is specified.
uint8_t	reserved2	Pad bytes reserved for future use and currently ignored.

Public Attribute Documentation

channel

```
uint16_t RAIL_RxChannelHoppingConfigEntry_t::channel
```

The channel number to be used for this entry in the channel hopping sequence.

If this is an invalid channel for the current PHY, the call to [RAIL_ConfigRxChannelHopping\(\)](#) will fail.

Definition at line 5336 of file `common/rail_types.h`

mode

```
RAIL_RxChannelHoppingMode_t RAIL_RxChannelHoppingConfigEntry_t::mode
```

The mode by which RAIL determines when to hop to the next channel.

Definition at line 5338 of file `common/rail_types.h`

parameter

```
RAIL_RxChannelHoppingParameter_t RAIL_RxChannelHoppingConfigEntry_t::parameter
```

Depending on the 'mode' parameter that was specified, this member is used to parameterize that mode.

See the comments on each value of [RAIL_RxChannelHoppingMode_t](#) to learn what to specify here.

Definition at line 5345 of file `common/rail_types.h`

delay

```
uint32_t RAIL_RxChannelHoppingConfigEntry_t::delay
```

Idle time in microseconds to wait before hopping into the channel indicated by this entry.

Definition at line 5350 of file `common/rail_types.h`

delayMode

```
RAIL_RxChannelHoppingDelayMode_t RAIL_RxChannelHoppingConfigEntry_t::delayMode
```

DeprecatedSet delayMode to [RAIL_RX_CHANNEL_HOPPING_DELAY_MODE_STATIC](#).

Definition at line 5352 of file `common/rail_types.h`

options

```
RAIL_RxChannelHoppingOptions_t RAIL_RxChannelHoppingConfigEntry_t::options
```

Bitmask of various options that can be applied to the current channel hop.

Definition at line 5357 of file `common/rail_types.h`

rssThresholdDbm

```
int8_t RAIL_RxChannelHoppingConfigEntry_t::rssThresholdDbm
```

The RSSI threshold (in dBm) below which a hop will occur in any mode when [RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD](#) is specified.

Definition at line 5363 of file `common/rail_types.h`

reserved2

```
uint8_t RAIL_RxChannelHoppingConfigEntry_t::reserved2[1]
```

Pad bytes reserved for future use and currently ignored.

Definition at line 5367 of file common/rail_types.h

RAIL_RxChannelHoppingConfig_t

Wrapper struct that will contain the sequence of [RAIL_RxChannelHoppingConfigEntry_t](#) that represents the channel sequence to use during RX Channel Hopping.

Public Attributes

<code>uint32_t *</code>	buffer	Pointer to contiguous global read-write memory that will be used by RAIL to store channel hopping information throughout its operation.
<code>uint16_t</code>	bufferLength	This parameter must be set to the length of the buffer array, in 32 bit words.
<code>uint8_t</code>	numberOfChannels	The number of channels in the channel hopping sequence, which is the number of elements in the array that entries points to.
RAIL_RxChannelHoppingConfigEntry_t *	entries	A pointer to the first element of an array of RAIL_RxChannelHoppingConfigEntry_t that represents the channels used during channel hopping.

Public Attribute Documentation

buffer

```
uint32_t* RAIL_RxChannelHoppingConfig_t::buffer
```

Pointer to contiguous global read-write memory that will be used by RAIL to store channel hopping information throughout its operation.

It need not be initialized and applications should never write data anywhere in this buffer.

Note

- The size of this buffer must be at least as large as $3 + \text{RAIL\_CHANNEL\_HOPPING\_BUFFER\_SIZE\_PER\_CHANNEL} * \text{numberOfChannels}$, plus the sum of the sizes of the `radioConfigDeltaAdd`'s of the required channels, plus the size of the `radioConfigDeltaSubtract`. In the case that one channel appears two or more times in your channel sequence (e.g., 1, 2, 3, 2), you must account for the radio configuration size that number of times (i.e., need to count channel 2's radio configuration size twice for the given example). The buffer is for internal use to the library.

Definition at line 5394 of file `common/rail_types.h`

bufferLength

```
uint16_t RAIL_RxChannelHoppingConfig_t::bufferLength
```

This parameter must be set to the length of the buffer array, in 32 bit words.

This way, during configuration, the software can confirm it's writing within the bounds of the buffer. The configuration API will return an error or trigger [RAIL_ASSERT_CHANNEL_HOPPING_BUFFER_TOO_SHORT](#) if `bufferLength` is insufficient.

Definition at line 5402 of file `common/rail_types.h`

numberOfChannels

```
uint8_t RAIL_RxChannelHoppingConfig_t::numberOfChannels
```

The number of channels in the channel hopping sequence, which is the number of elements in the array that entries points to.

Definition at line 5407 of file `common/rail_types.h`

entries

```
RAIL_RxChannelHoppingConfigEntry_t* RAIL_RxChannelHoppingConfig_t::entries
```

A pointer to the first element of an array of [RAIL_RxChannelHoppingConfigEntry_t](#) that represents the channels used during channel hopping.

This array must have numberOfChannels entries.

Definition at line 5414 of file `common/rail_types.h`

RAIL_RxDutyCycleConfig_t

Structure to configure duty cycled receive mode.

Public Attributes

RAIL_RxChannelHoppingMode_t	mode The mode by which RAIL determines when to exit RX.
RAIL_RxChannelHoppingParameter_t	parameter Depending on the 'mode' parameter that was specified, this member is used to parameterize that mode.
uint32_t	delay Idle time in microseconds to wait before re-entering RX.
RAIL_RxChannelHoppingDelayMode_t	delayMode
RAIL_RxChannelHoppingOptions_t	options Bitmask of various options that can be applied to the current duty cycle operation when the mode is >= RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL_WITH_OPTIONS (ignored otherwise).
int8_t	rssiThresholdDbm The RSSI threshold (in dBm) below which Rx will end in any mode when RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD is specified.
uint8_t	reserved2 Pad bytes reserved for future use and currently ignored.

Public Attribute Documentation

mode

```
RAIL_RxChannelHoppingMode_t RAIL_RxDutyCycleConfig_t::mode
```

The mode by which RAIL determines when to exit RX.

Definition at line 5423 of file `common/rail_types.h`

parameter

```
RAIL_RxChannelHoppingParameter_t RAIL_RxDutyCycleConfig_t::parameter
```

Depending on the 'mode' parameter that was specified, this member is used to parameterize that mode.

See the comments on each value of [RAIL_RxChannelHoppingMode_t](#) to learn what to specify here.

Definition at line 5430 of file `common/rail_types.h`

delay


```
uint32_t RAIL_RxDutyCycleConfig_t::delay
```

Idle time in microseconds to wait before re-entering RX.

Definition at line 5434 of file common/rail_types.h

delayMode

```
RAIL_RxChannelHoppingDelayMode_t RAIL_RxDutyCycleConfig_t::delayMode
```

DeprecatedSet delayMode to [RAIL_RX_CHANNEL_HOPPING_DELAY_MODE_STATIC](#).

Definition at line 5436 of file common/rail_types.h

options

```
RAIL_RxChannelHoppingOptions_t RAIL_RxDutyCycleConfig_t::options
```

Bitmask of various options that can be applied to the current duty cycle operation when the mode is >= [RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL_WITH_OPTIONS](#) (ignored otherwise).

Definition at line 5442 of file common/rail_types.h

rssThresholdDbm

```
int8_t RAIL_RxDutyCycleConfig_t::rssThresholdDbm
```

The RSSI threshold (in dBm) below which Rx will end in any mode when [RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD](#) is specified.

Definition at line 5448 of file common/rail_types.h

reserved2

```
uint8_t RAIL_RxDutyCycleConfig_t::reserved2[1]
```

Pad bytes reserved for future use and currently ignored.

Definition at line 5452 of file common/rail_types.h

EFR32xG2x

EFR32xG2x

EFR32xG2x-specific RX channel hopping.

Macros

#define

RAIL_CHANNEL_HOPPING_BUFFER_SIZE_PER_CHANNEL (56U)

The static amount of memory needed per channel for channel hopping, measured in 32 bit words, regardless of the size of radio configuration structures.

Slxx3xx

Slxx3xx

Slxx3xx-specific RX channel hopping.

Macros

#define	RAIL_CHANNEL_HOPPING_BUFFER_SIZE_PER_CHANNEL (54U)
	The static amount of memory needed per channel for channel hopping, measured in 32 bit words, regardless of the size of radio configuration structures.

Radio Configuration

Radio Configuration

Routines for setting up and querying radio configuration information.

These routines allow for runtime flexibility in the radio configuration. Some of the parameters, however, are meant to be generated from the radio calculator in Simplicity Studio. The basic code to configure the radio from this calculator output looks like the example below.

```
// Associate a specific channel configuration with a particular RAIL instance and
// load the settings that correspond to the first usable channel.
RAIL_ConfigChannels(railHandle, channelConfigs[0]);
```

For more information about the types of parameters that can be changed in the other functions and how to use them, see their individual documentation.

Modules

[RAIL_FrameType_t](#)

[RAIL_AlternatePhy_t](#)

[RAIL_ChannelConfigEntry_t](#)

[RAIL_ChannelConfig_t](#)

[RAIL_ChannelMetadata_t](#)

[RAIL_StackInfoCommon_t](#)

[RAIL_ChannelConfigEntryAttr_t](#)

Enumerations

```
enum    RAIL_ChannelConfigEntryType_t {
        RAIL_CH_TYPE_NORMAL
        RAIL_CH_TYPE_CONC_BASE
        RAIL_CH_TYPE_CONC_VIRTUAL
    }
```

Define if the channel support using concurrent PHY during channel hopping.

Typedefs

```
typedef const uint32_t *    RAIL_RadioConfig_t
                             Pointer to a radio configuration array.
```

```
typedef void(*)            RAIL_RadioConfigChangedCallback_t)(RAIL_Handle_t railHandle, const RAIL_ChannelConfigEntry_t *entry)
                             A pointer to a function called whenever a radio configuration change occurs.
```

Functions

uint16_t	RAIL_SetFixedLength (RAIL_Handle_t railHandle, uint16_t length) Modify the currently configured fixed frame length in bytes.
uint16_t	RAIL_ConfigChannels (RAIL_Handle_t railHandle, const RAIL_ChannelConfig_t *config, RAIL_RadioConfigChangedCallback_t cb) Configure the channels supported by this device.
RAIL_Status_t	RAIL_ConfigChannelsAlt (RAIL_Handle_t railHandle, const RAIL_ChannelConfig_t *config, RAIL_RadioConfigChangedCallback_t cb) Configure the channels supported by this device.
RAIL_Status_t	RAIL_GetChannelMetadata (RAIL_Handle_t railHandle, RAIL_ChannelMetadata_t *channelMetadata, uint16_t *length, uint16_t minChannel, uint16_t maxChannel) Get verbose listing of channel metadata for the current channel configuration.
RAIL_Status_t	RAIL_IsValidChannel (RAIL_Handle_t railHandle, uint16_t channel) Check whether the channel exists in the current RAIL channel configuration.
RAIL_Status_t	RAIL_PrepareChannel (RAIL_Handle_t railHandle, uint16_t channel) Cause radio settings associated with a particular channel to be applied to hardware.
RAIL_Status_t	RAIL_GetChannel (RAIL_Handle_t railHandle, uint16_t *channel) Return the most-recently requested RAIL channel.
RAIL_Status_t	RAIL_GetChannelAlt (RAIL_Handle_t railHandle, uint16_t *channel) Return the RAIL channel to which the radio is currently tuned.
uint32_t	RAIL_GetSymbolRate (RAIL_Handle_t railHandle) Return the symbol rate for the current PHY.
uint32_t	RAIL_GetBitRate (RAIL_Handle_t railHandle) Return the bit rate for the current PHY.
RAIL_Status_t	RAIL_SetPaCtune (RAIL_Handle_t railHandle, uint8_t txPaCtuneValue, uint8_t rxPaCtuneValue) Set the PA capacitor tune value for transmit and receive.
RAIL_Status_t	RAIL_GetSyncWords (RAIL_Handle_t railHandle, RAIL_SyncWordConfig_t *syncWordConfig) Get the sync words and their length.
RAIL_Status_t	RAIL_ConfigSyncWords (RAIL_Handle_t railHandle, const RAIL_SyncWordConfig_t *syncWordConfig) Set the selected sync words and their length.
uint16_t	RAIL_GetWhiteningInitVal (RAIL_Handle_t railHandle) Sets the whitening initialization value.
uint32_t	RAIL_GetCrclnitVal (RAIL_Handle_t railHandle) Returns the CRC initialization value.
RAIL_Status_t	RAIL_SetWhiteningInitVal (RAIL_Handle_t railHandle, uint16_t whiteInit) Sets the whitening initialization value.
RAIL_Status_t	RAIL_SetCrclnitVal (RAIL_Handle_t railHandle, uint32_t crclnit) Sets the CRC initialization value.
RAIL_Status_t	RAIL_ResetWhiteningInitVal (RAIL_Handle_t railHandle) Restores the whitening initialization value to its initial setting from the Radio Configurator.
RAIL_Status_t	RAIL_ResetCrclnitVal (RAIL_Handle_t railHandle) Restores the CRC initialization value to its initial setting from the Radio Configurator.

Macros

- #define RAIL_SETFIXEDLENGTH_INVALID (0xFFFFU)
An invalid return value when calling [RAIL_SetFixedLength\(\)](#).
- #define RADIO_CONFIG_ENABLE_CONC_PHY 1
Indicates this version of RAIL supports concurrent PHY information in radio configurator output.
- #define RADIO_CONFIG_ENABLE_STACK_INFO
Indicates this version of RAIL supports stack info feature in radio configurator output.

Enumeration Documentation

RAIL_ChannelConfigEntryType_t

RAIL_ChannelConfigEntryType_t

Define if the channel support using concurrent PHY during channel hopping.

[RAIL_RX_CHANNEL_HOPPING_MODE_CONC](#) and [RAIL_RX_CHANNEL_HOPPING_MODE_VT](#) can only be used if the channel supports it.

Enumerator	
RAIL_CH_TYPE_NORMAL	Not a concurrent PHY.
RAIL_CH_TYPE_CONC_BASE	Base concurrent PHY.
RAIL_CH_TYPE_CONC_VIRTUAL	Virtual concurrent PHY.

Definition at line 2039 of file `common/railTypes.h`

Typedef Documentation

RAIL_RadioConfig_t

typedef const uint32_t* RAIL_RadioConfig_t

Pointer to a radio configuration array.

The radio configuration properly configures the radio for operation on a protocol. These configurations are very chip-specific should not be created or edited by hand.

Definition at line 1976 of file `common/railTypes.h`

RAIL_RadioConfigChangedCallback_t

RAIL_RadioConfigChangedCallback_t (RAIL_Handle_t railHandle, const RAIL_ChannelConfigEntry_t *entry)

A pointer to a function called whenever a radio configuration change occurs.

Parameters

[in]	railHandle	The corresponding RAIL instance handle.
[in]	entry	A pointer to the radio configuration entry being changed to.

Definition at line 2403 of file `common/railTypes.h`

Function Documentation

RAIL_SetFixedLength

```
uint16_t RAIL_SetFixedLength (RAIL_Handle_t railHandle, uint16_t length)
```

Modify the currently configured fixed frame length in bytes.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	length	The expected fixed frame length. A value of 0 is infinite. A value of RAIL_SETFIXEDLENGTH_INVALID restores the frame's length back to the length specified by the default frame type configuration.

Returns

- The new frame length configured into the hardware for use: 0 if in infinite mode, or [RAIL_SETFIXEDLENGTH_INVALID](#) if the frame length has not yet been overridden by a valid value.

Sets the fixed-length configuration for transmit and receive. Be careful when using this function in receive and transmit as this function changes the default frame configuration and remains in force until it is called again with an input value of [RAIL_SETFIXEDLENGTH_INVALID](#). This function will override any fixed or variable length settings from a radio configuration.

Definition at line 609 of file common/rail.h

RAIL_ConfigChannels

```
uint16_t RAIL_ConfigChannels (RAIL_Handle_t railHandle, const RAIL_ChannelConfig_t * config, RAIL_RadioConfigChangedCallback_t cb)
```

Configure the channels supported by this device.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	A pointer to the channel configuration for your device. This pointer will be cached in the library so it must exist for the runtime of the application. Typically, this should be what is stored in Flash by the configuration tool.
[in]	cb	A pointer to a function called whenever a radio configuration change occurs. May be NULL if do not need a callback.

Returns

- The first available channel in the configuration.

When configuring channels on EFR32, the radio tuner is reconfigured based on the frequency and channel spacing in the channel configuration and the first channel in the configuration is implicitly prepared as if [RAIL_PrepareChannel\(\)](#) were called.

Note

- config can be NULL to simply register or unregister the cb callback function when using RAIL internal protocol-specific radio configuration APIs for BLE, IEEE 802.15.4, or Z-Wave, which lack callback specification. In this use case, 0 is returned.

Definition at line 633 of file common/rail.h

RAIL_ConfigChannelsAlt

```
RAIL_Status_t RAIL_ConfigChannelsAlt (RAIL_Handle_t railHandle, const RAIL_ChannelConfig_t * config,
RAIL_RadioConfigChangedCallback_t cb)
```

Configure the channels supported by this device.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	A pointer to the channel configuration for your device. This pointer will be cached in the library so it must exist for the runtime of the application. Typically, this should be what is stored in Flash by the configuration tool.
[in]	cb	A pointer to a function called whenever a radio configuration change occurs. May be NULL if do not need a callback.

Returns

- Status code indicating success of the function call.

Note

- Unlike [RAIL_ConfigChannels\(\)](#), this function only caches the configuration and does not prepare any channel in the configuration. That action is deferred to the next call to a RAIL API where channel is passed as a parameter, namely [RAIL_PrepareChannel\(\)](#), [RAIL_StartTx\(\)](#), [RAIL_StartScheduledTx\(\)](#), [RAIL_StartCcaCsmatx\(\)](#), [RAIL_StartCcaLbtTx\(\)](#), [RAIL_StartScheduledCcaCsmatx\(\)](#), [RAIL_StartScheduledCcaLbtTx\(\)](#), [RAIL_StartRx\(\)](#), [RAIL_ScheduleRx\(\)](#), [RAIL_StartAverageRssi\(\)](#), [RAIL_StartTxStream\(\)](#).
- config can be NULL to simply register or unregister the cb callback function when using RAIL internal protocol-specific radio configuration APIs for BLE, IEEE 802.15.4, or Z-Wave, which lack callback specification.

Definition at line 664 of file common/rail.h

RAIL_GetChannelMetadata

```
RAIL_Status_t RAIL_GetChannelMetadata (RAIL_Handle_t railHandle, RAIL_ChannelMetadata_t * channelMetadata, uint16_t
* length, uint16_t minChannel, uint16_t maxChannel)
```

Get verbose listing of channel metadata for the current channel configuration.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	channelMetadata	A pointer to an app-allocated array that will be populated with channel metadata.
[inout]	length	A pointer to the number of entries available in the channelMetadata array. This value will be updated to the number of channels written to the array, not to exceed the length number passed in.
[in]	minChannel	Minimum channel number about which to collect data.
[in]	maxChannel	Maximum channel number about which to collect data.

Returns

- Status code indicating success of the function call. [RAIL_STATUS_INVALID_PARAMETER](#) means that, based on the currently active radio configuration, there are more channels to write than there is space provided in the allocated channelMetadata. However, the channel metadata that was written is valid. [RAIL_STATUS_INVALID_STATE](#) indicates that the channel configuration has not been configured. [RAIL_STATUS_NO_ERROR](#) indicates complete success.

Definition at line 689 of file common/rail.h

RAIL_IsValidChannel

```
RAIL_Status_t RAIL_IsValidChannel (RAIL_Handle_t railHandle, uint16_t channel)
```

Check whether the channel exists in the current RAIL channel configuration.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	A channel number to check.

Returns

- [RAIL_STATUS_NO_ERROR](#) if channel exists or [RAIL_STATUS_INVALID_PARAMETER](#) if the given channel does not exist.

Definition at line 703 of file common/rail.h

RAIL_PrepareChannel

```
RAIL_Status_t RAIL_PrepareChannel (RAIL_Handle_t railHandle, uint16_t channel)
```

Cause radio settings associated with a particular channel to be applied to hardware.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	The channel to prepare for use.

Returns

- [RAIL_STATUS_NO_ERROR](#) on success or [RAIL_STATUS_INVALID_PARAMETER](#) if the given channel does not have an associated channel configuration entry.

This function walks the current [RAIL_ChannelConfig_t::configs](#) list and applies the configuration associated with the specified channel if found. This function manually changes channels without starting a TX or RX operation.

When successful, the radio is idled. When unsuccessful, the radio state will not be altered.

Definition at line 725 of file common/rail.h

RAIL_GetChannel

```
RAIL_Status_t RAIL_GetChannel (RAIL_Handle_t railHandle, uint16_t * channel)
```

Return the most-recently requested RAIL channel.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	channel	A pointer to the channel for which RAIL was most-recently configured.

Returns

- [RAIL_STATUS_NO_ERROR](#) on success or [RAIL_STATUS_INVALID_CALL](#) if the radio is not configured for any channel or [RAIL_STATUS_INVALID_PARAMETER](#) if channel parameter is NULL.

This function returns the channel most recently specified in API calls that pass in a channel to tune to, namely [RAIL_PrepareChannel\(\)](#), [RAIL_StartTx\(\)](#), [RAIL_StartScheduledTx\(\)](#), [RAIL_StartCcaCsmaTx\(\)](#), [RAIL_StartCcaLbtTx\(\)](#), [RAIL_StartScheduledCcaCsmaTx\(\)](#), [RAIL_StartScheduledCcaLbtTx\(\)](#), [RAIL_StartRx\(\)](#), [RAIL_ScheduleRx\(\)](#), [RAIL_StartAverageRssi\(\)](#), [RAIL_StartTxStream\(\)](#), [RAIL_StartTxStreamAlt\(\)](#). It doesn't follow changes RAIL performs implicitly during channel hopping and mode switch.

Definition at line 746 of file `common/rail.h`

RAIL_GetChannelAlt

```
RAIL_Status_t RAIL_GetChannelAlt (RAIL_Handle_t railHandle, uint16_t * channel)
```

Return the RAIL channel to which the radio is currently tuned.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	channel	A pointer to the currently-tuned channel.

Returns

- [RAIL_STATUS_NO_ERROR](#) on success or [RAIL_STATUS_INVALID_CALL](#) if the radio is not configured for any channel or [RAIL_STATUS_INVALID_PARAMETER](#) if channel parameter is NULL.

This function returns the channel to which the radio is currently tuned if the specified RAIL handle is active. It returns the channel to which it will be tuned during the next protocol switch if the handle is inactive. The channel returned may be different than what [RAIL_GetChannel](#) returns when channel hopping or mode switch are involved.

Definition at line 763 of file `common/rail.h`

RAIL_GetSymbolRate

```
uint32_t RAIL_GetSymbolRate (RAIL_Handle_t railHandle)
```

Return the symbol rate for the current PHY.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The symbol rate in symbols per second or 0.

The symbol rate is the rate of symbol changes over the air. For non-DSSS PHYs, this is the same as the baudrate. For DSSS PHYs, it is the baudrate divided by the length of a chipping sequence. For more information, see the modem calculator documentation. If the rate cannot be calculated, this function returns 0.

Definition at line 777 of file `common/rail.h`

RAIL_GetBitRate

```
uint32_t RAIL_GetBitRate (RAIL_Handle_t railHandle)
```

Return the bit rate for the current PHY.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The bit rate in bits per second or 0.

The bit rate is the effective over-the-air data rate. It does not account for extra spreading for forward error correction, and so on, but accounts for modulation schemes, DSSS, and other configurations. For more information, see the modem calculator documentation. If the rate cannot be calculated, this function returns 0.

Definition at line 805 of file common/rail.h

RAIL_SetPaCTune

```
RAIL_Status_t RAIL_SetPaCTune (RAIL_Handle_t railHandle, uint8_t txPaCtuneValue, uint8_t rxPaCtuneValue)
```

Set the PA capacitor tune value for transmit and receive.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	txPaCtuneValue	PA Ctune value for TX mode.
[in]	rxPaCtuneValue	PA Ctune value for RX mode.

Returns

- Status code indicating success of the function call.

Tunes the impedance of the transmit and receive modes by changing the amount of capacitance at the PA output. Changes made to the TX Power configuration, e.g., calling [RAIL_ConfigTxPower](#), will undo changes made to PA capacitor tune value for transmit and receive via [RAIL_SetPaCTune](#).

Note

- This function does nothing on EFR32 Series 2 devices.

Definition at line 838 of file common/rail.h

RAIL_GetSyncWords

```
RAIL_Status_t RAIL_GetSyncWords (RAIL_Handle_t railHandle, RAIL_SyncWordConfig_t * syncWordConfig)
```

Get the sync words and their length.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	syncWordConfig	An application-provided non-NULL pointer to store RAIL_SyncWordConfig_t sync word information.

Returns

- Status code indicating success of the function call.

Definition at line 850 of file common/rail.h

RAIL_ConfigSyncWords

```
RAIL_Status_t RAIL_ConfigSyncWords (RAIL_Handle_t railHandle, const RAIL_SyncWordConfig_t * syncWordConfig)
```

Set the selected sync words and their length.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	syncWordConfig	A non-NULL pointer to RAIL_SyncWordConfig_t specifying the sync words and their length. The desired length should be between 2 and 32 bits inclusive, however it is recommended to not change the length below what the PHY sync word length is configured to be. Changing the sync word length, especially to that which is lower than the default length, may result in a decrease in packet reception rate or may not work at all. Other values will result in RAIL_STATUS_INVALID_PARAMETER . The default sync word continues to be valid.

Returns

- Status code indicating success of the function call.

When the custom sync word(s) applied by this API are no longer needed, or to revert to default sync word, calling [RAIL_ConfigChannels\(\)](#) will re-establish the sync words specified in the radio configuration.

This function will return [RAIL_STATUS_INVALID_STATE](#) if called when BLE has been enabled for this railHandle. When changing sync words in BLE mode, use [RAIL_BLE_ConfigChannelRadioParams\(\)](#) instead.

Note

- If multiple protocols share the same radio configuration, the user should not set custom sync words in any of those protocols as these sync words could leak into the other protocol sharing the same radio configuration.

Definition at line 881 of file `common/rail.h`

RAIL_GetWhiteningInitVal

```
uint16_t RAIL_GetWhiteningInitVal (RAIL_Handle_t railHandle)
```

Sets the whitening initialization value.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The whitening initialization value currently being used.

Definition at line 890 of file `common/rail.h`

RAIL_GetCrclnitVal

```
uint32_t RAIL_GetCrclnitVal (RAIL_Handle_t railHandle)
```

Returns the CRC initialization value.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The CRC initialization value currently being used.

Definition at line 898 of file common/rail.h

RAIL_SetWhiteningInitVal

```
RAIL_Status_t RAIL_SetWhiteningInitVal (RAIL_Handle_t railHandle, uint16_t whitelnit)
```

Sets the whitening initialization value.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	whitelnit	A whitening initialization value.

Returns

- Status code indicating success of the function call.

Use this function to override the whitening initialization value defined by the current PHY's radio configuration. The new value will persist until this function is called again, [RAIL_ResetWhiteningInitVal\(\)](#) is called, or the PHY is changed.

Note

- Overriding a PHY's whitening initialization value will break communication with peers unless they effect a similar change.

Warnings

- This API must not be used when either 802.15.4 or BLE modes are enabled.

Definition at line 920 of file common/rail.h

RAIL_SetCrclnitVal

```
RAIL_Status_t RAIL_SetCrclnitVal (RAIL_Handle_t railHandle, uint32_t crclnit)
```

Sets the CRC initialization value.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	crclnit	A CRC initialization value.

Returns

- Status code indicating success of the function call.

Use this function to override the CRC initialization value defined by the current PHY's radio configuration. The new value will persist until this function is called again, [RAIL_ResetCrclnitVal\(\)](#) is called, or the PHY is changed.

Note

- Overriding a PHY's CRC initialization value will break communication with peers unless they effect a similar change.

Warnings

This API must not be used when either 802.15.4 or BLE modes are enabled.

Definition at line 942 of file common/rail.h

RAIL_ResetWhiteningInitVal

RAIL_Status_t RAIL_ResetWhiteningInitVal ([RAIL_Handle_t](#) railHandle)

Restores the whitening initialization value to its initial setting from the Radio Configurator.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.
- Can use this function after using [RAIL_SetWhiteningInitVal\(\)](#).

Definition at line 954 of file common/rail.h

RAIL_ResetCrclnitVal

RAIL_Status_t RAIL_ResetCrclnitVal ([RAIL_Handle_t](#) railHandle)

Restores the CRC initialization value to its initial setting from the Radio Configurator.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.
- Can use this function after using [RAIL_SetCrclnitVal\(\)](#).

Definition at line 965 of file common/rail.h

RAIL_FrameType_t

Configures if there is a frame type in your frame and the lengths of each frame.

The number of bits set in the mask determines the number of elements in frameLen. A maximum of 8 different frame types may be specified.

Public Attributes

uint16_t *	frameLen	A pointer to an array of frame byte lengths for each frame type.
uint8_t	offset	Zero-indexed byte offset location of the byte containing the frame type field.
uint8_t	mask	A bitmask of the frame type field, which determines a number of frames expected based on the number of bits set.
uint8_t	isValid	A bitmask that marks if each frame is valid or should be filtered.
uint8_t	addressFilter	A bitmask that marks if each frame should have the address filter applied.

Public Attribute Documentation

frameLen

```
uint16_t* RAIL_FrameType_t::frameLen
```

A pointer to an array of frame byte lengths for each frame type.

The number of elements in this array should be equal to the number of frame types. The memory to which frameLen points should not change location or be modified.

Definition at line 1991 of file common/rail_types.h

offset

```
uint8_t RAIL_FrameType_t::offset
```

Zero-indexed byte offset location of the byte containing the frame type field.

Definition at line 1995 of file common/rail_types.h

mask

```
uint8_t RAIL_FrameType_t::mask
```

A bitmask of the frame type field, which determines a number of frames expected based on the number of bits set.

No more than 3 bits can be set in the mask and they must be contiguous ones. For example, if the highest three bits of the byte specified by offset constitute the frame type, then mask should be 0xE0, which has 3 bits set, indicating 8 possible frame types.

Definition at line 2003 of file `common/rail_types.h`

isValid

```
uint8_t RAIL_FrameType_t::isValid
```

A bitmask that marks if each frame is valid or should be filtered.

Frame type 0 corresponds to the lowest bit in isValid. If the frame is filtered, a [RAIL_EVENT_RX_PACKET_ABORTED](#) will be raised.

Definition at line 2009 of file `common/rail_types.h`

addressFilter

```
uint8_t RAIL_FrameType_t::addressFilter
```

A bitmask that marks if each frame should have the address filter applied.

Frame type 0 corresponds to the least significant bit in addressFilter.

Definition at line 2014 of file `common/rail_types.h`

RAIL_AlternatePhy_t

Alternate PHY configuration entry structure, which gathers some info on the alternate PHY in the context of concurrent mode.

Public Attributes

uint32_t	baseFrequency	A base frequency in Hz of this channel set.
uint32_t	channelSpacing	A channel spacing in Hz of this channel set.
uint16_t	numberOfChannels	The number of channels (and not the channel number !)
uint16_t	minIf_kHz	minimum IF for the alternate PHY in kHz.
uint16_t	minBaseIf_kHz	minimum IF for the base PHY in kHz.
bool	isOfdmModem	Indicates that OFDM modem is used by this alternate PHY.
uint32_t	rateInfo	Rate info of the alternate PHY.
uint32_t	hwModemAgcCtrl	Used to adjust the AGC for CCA between hard and soft modems.

Public Attribute Documentation

baseFrequency

uint32_t RAIL_AlternatePhy_t::baseFrequency

A base frequency in Hz of this channel set.

Definition at line 2076 of file common/rail_types.h

channelSpacing

uint32_t RAIL_AlternatePhy_t::channelSpacing

A channel spacing in Hz of this channel set.

Definition at line 2078 of file common/rail_types.h

numberOfChannels

uint16_t RAIL_AlternatePhy_t::numberOfChannels

The number of channels (and not the channel number !)

Definition at line 2080 of file common/rail_types.h

minlf_kHz

uint16_t RAIL_AlternatePhy_t::minlf_kHz

minimum IF for the alternate PHY in kHz.

Definition at line 2082 of file common/rail_types.h

minBaself_kHz

uint16_t RAIL_AlternatePhy_t::minBaself_kHz

minimum IF for the base PHY in kHz.

Definition at line 2084 of file common/rail_types.h

isOfdmModem

bool RAIL_AlternatePhy_t::isOfdmModem

Indicates that OFDM modem is used by this alternate PHY.

Definition at line 2086 of file common/rail_types.h

rateInfo

uint32_t RAIL_AlternatePhy_t::rateInfo

Rate info of the alternate PHY.

Definition at line 2088 of file common/rail_types.h

hwModemAgcCtrl1

uint32_t RAIL_AlternatePhy_t::hwModemAgcCtrl1

Used to adjust the AGC for CCA between hard and soft modems.

Definition at line 2090 of file common/rail_types.h

RAIL_ChannelConfigEntry_t

A channel configuration entry structure, which defines a channel range and parameters across which a corresponding radio configuration is valid.

operating frequency hz = baseFrequency

- channelSpacing * (channel - physicalChannelOffset);

Public Attributes

RAIL_RadioConfig_t	phyConfigDeltaAdd	The minimum radio configuration to apply to the base configuration for this channel set.
uint32_t	baseFrequency	A base frequency in Hz of this channel set.
uint32_t	channelSpacing	A channel spacing in Hz of this channel set.
uint16_t	physicalChannelOffset	The offset to subtract from the logical channel to align them with the zero based physical channels which are relative to baseFrequency.
uint16_t	channelNumberStart	The first valid RAIL channel number for this channel set.
uint16_t	channelNumberEnd	The last valid RAIL channel number for this channel set.
RAIL_TxPower_t	maxPower	The maximum power allowed in this channel set.
RAIL_ChannelConfigEntryAttr_t *	attr	A pointer to a structure containing attributes specific to this channel set.
RAIL_ChannelConfigEntryType_t	entryType	Indicates channel config type.
uint8_t	reserved	to align to 32-bit boundary.
const uint8_t *	stackInfo	Array containing information according to the RAIL_PtiProtocol_t in the first byte of this array.
RAIL_AlternatePhy_t *	alternatePhy	Pointer to alternate PHY.

Public Attribute Documentation

phyConfigDeltaAdd

RAIL_RadioConfig_t RAIL_ChannelConfigEntry_t::phyConfigDeltaAdd

The minimum radio configuration to apply to the base configuration for this channel set.

Definition at line 2106 of file common/rail_types.h

baseFrequency

```
uint32_t RAIL_ChannelConfigEntry_t::baseFrequency
```

A base frequency in Hz of this channel set.

Definition at line 2108 of file common/rail_types.h

channelSpacing

```
uint32_t RAIL_ChannelConfigEntry_t::channelSpacing
```

A channel spacing in Hz of this channel set.

Definition at line 2110 of file common/rail_types.h

physicalChannelOffset

```
uint16_t RAIL_ChannelConfigEntry_t::physicalChannelOffset
```

The offset to subtract from the logical channel to align them with the zero based physical channels which are relative to baseFrequency.

(i.e., By default ch 0 = base freq, but if offset = 11, ch 11 = base freq.)

Definition at line 2117 of file common/rail_types.h

channelNumberStart

```
uint16_t RAIL_ChannelConfigEntry_t::channelNumberStart
```

The first valid RAIL channel number for this channel set.

Definition at line 2119 of file common/rail_types.h

channelNumberEnd

```
uint16_t RAIL_ChannelConfigEntry_t::channelNumberEnd
```

The last valid RAIL channel number for this channel set.

Definition at line 2121 of file common/rail_types.h

maxPower

```
RAIL_TxPower_t RAIL_ChannelConfigEntry_t::maxPower
```

The maximum power allowed in this channel set.

Definition at line 2123 of file common/rail_types.h

attr

```
RAIL_ChannelConfigEntryAttr_t* RAIL_ChannelConfigEntry_t::attr
```

A pointer to a structure containing attributes specific to this channel set.

Definition at line 2128 of file common/rail_types.h

entryType

```
RAIL_ChannelConfigEntryType_t RAIL_ChannelConfigEntry_t::entryType
```

Indicates channel config type.

Definition at line 2130 of file common/rail_types.h

reserved

```
uint8_t RAIL_ChannelConfigEntry_t::reserved[3]
```

to align to 32-bit boundary.

Definition at line 2132 of file common/rail_types.h

stackInfo

```
const uint8_t* RAIL_ChannelConfigEntry_t::stackInfo
```

Array containing information according to the [RAIL_PtiProtocol_t](#) in the first byte of this array.

The first 2 fields are common to all protocols and accessible by RAIL, others are ignored by RAIL and only used by the application. Common fields are listed in [RAIL_StackInfoCommon_t](#).

Definition at line 2140 of file common/rail_types.h

alternatePhy

```
RAIL_AlternatePhy_t* RAIL_ChannelConfigEntry_t::alternatePhy
```

Pointer to alternate PHY.

Definition at line 2142 of file common/rail_types.h

RAIL_ChannelConfig_t

A channel configuration structure, which defines the channel meaning when a channel number is passed into a RAIL function, e.g., [RAIL_StartTx\(\)](#) and [RAIL_StartRx\(\)](#).

A [RAIL_ChannelConfig_t](#) structure defines the channel scheme that an application uses when registered in [RAIL_ConfigChannels\(\)](#).

These are a few examples of different channel configurations:

```

// 21 channels starting at 2.45 GHz with channel spacing of 1 MHz
// ... generated by Simplicity Studio (i.e., rail_config.c) ...
const uint32_t generated[] = { ... };
RAIL_ChannelConfigEntryAttr_t generated_entryAttr = { ... };
const RAIL_ChannelConfigEntry_t generated_channels[] = {
{
    .phyConfigDeltaAdd = NULL, // Add this to default configuration for this entry
    .baseFrequency = 245000000,
    .channelSpacing = 1000000,
    .physicalChannelOffset = 0,
    .channelNumberStart = 0,
    .channelNumberEnd = 20,
    .maxPower = RAIL_TX_POWER_MAX,
    .attr = &generated_entryAttr,
},
};

const RAIL_ChannelConfig_t generated_channelConfig = {
    .phyConfigBase = generated, // Default radio configuration for all entries
    .phyConfigDeltaSubtract = NULL, // Subtract this to restore the default configuration
    .configs = generated_channels,
    .length = 1, // There are this many channel configuration entries
};

const RAIL_ChannelConfig_t *channelConfigs[] = {
    &generated_channelConfig,
    NULL,
};

// ... in main code ...
// Associate a specific channel configuration with a particular RAIL instance.
RAIL_ConfigChannels(railHandle, channelConfigs[0]);

// 4 nonlinear channels
// ... in rail_config.c ...
const uint32_t generated[] = { ... };
RAIL_ChannelConfigEntryAttr_t generated_entryAttr = { ... };
const RAIL_ChannelConfigEntry_t generated_channels[] = {
{
    .phyConfigDeltaAdd = NULL, // Add this to default configuration for this entry
    .baseFrequency = 910123456,
    .channelSpacing = 0,
    .physicalChannelOffset = 0,
    .channelNumberStart = 0,
    .channelNumberEnd = 0,
    .maxPower = RAIL_TX_POWER_MAX,
    .attr = &generated_entryAttr,
},
{
    .phyConfigDeltaAdd = NULL,
    .baseFrequency = 911654789,
    .channelSpacing = 0,
    .physicalChannelOffset = 0, // Since ch spacing = 0, offset can be 0
    .channelNumberStart = 1,
    .channelNumberEnd = 1,
    .maxPower = RAIL_TX_POWER_MAX,
    .attr = &generated_entryAttr,
},
{
    .phyConfigDeltaAdd = NULL,
    .baseFrequency = 912321456,
    .channelSpacing = 100000,
    .physicalChannelOffset = 2, // Since ch spacing != 0, offset = 2
    .channelNumberStart = 2, // ch 2 = baseFrequency
    .channelNumberEnd = 2,
    .maxPower = RAIL_TX_POWER_MAX,
    .attr = &generated_entryAttr,
},
{

```



```

.baseFrequency = 913147852, channelSpacing = 0, physicalChannelOffset = 0, channelNumberStart = 3, channelNumberEnd = 3, maxPower =
RAIL_TX_POWER_MAX, attr = &generated_entryAttr,};};
const RAIL_ChannelConfig_t generated_channelConfig = { .phyConfigBase = generated, // Default radio configuration for all
entries.phyConfigDeltaSubtract = NULL, // Subtract this to restore the default configuration.configs = generated_channels, length = 4, // There are
this many channel configuration entries};
const RAIL_ChannelConfig_t *channelConfigs[] = { &generated_channelConfig,
NULL,}; // ... in main code ... // Associate a specific channel configuration with a particular RAIL instance.RAIL_ConfigChannels(railHandle,
channelConfigs[0]); // Multiple radio configurations // ... in rail_config.c ...
const uint32_t generated0[] = {...};
RAIL_ChannelConfigEntryAttr_t generated0_entryAttr = {...};
const RAIL_ChannelConfigEntry_t generated0_channels[] = { { .phyConfigDeltaAdd = NULL, // Add this to the default configuration for this
entry.baseFrequency = 2450000000, channelSpacing = 1000000, physicalChannelOffset = 0, channelNumberStart = 0, channelNumberEnd
= 20, maxPower = RAIL_TX_POWER_MAX, attr = &generated0_entryAttr,},};
const RAIL_ChannelConfig_t generated0_channelConfig = { .phyConfigBase = generated0, // Default radio configuration for all
entries.phyConfigDeltaSubtract = NULL, // Subtract this to restore default configuration.configs = generated0_channels, length = 1, // There are this
many channel configuration entries};
const uint32_t generated1[] = {...};
RAIL_ChannelConfigEntryAttr_t generated1_entryAttr = {...};
const RAIL_ChannelConfigEntry_t generated1_channels[] = { { .phyConfigDeltaAdd = NULL, baseFrequency = 2450000000, channelSpacing
= 1000000, physicalChannelOffset = 0, channelNumberStart = 0, channelNumberEnd = 20, maxPower = -100, // Use this entry when TX power <=
-10dBm, attr = &generated1_entryAttr,}, { .phyConfigDeltaAdd = NULL, baseFrequency = 2450000000, channelSpacing
= 1000000, physicalChannelOffset = 0, channelNumberStart = 0, channelNumberEnd = 20, maxPower = 15, // Use this entry when TX power >
-10dBm // and TX power <= 1.5dBm, attr = &generated1_entryAttr,}, { .phyConfigDeltaAdd = NULL, baseFrequency = 2450000000, channelSpacing
= 1000000, physicalChannelOffset = 0, channelNumberStart = 0, channelNumberEnd = 20, maxPower = RAIL_TX_POWER_MAX, // Use this entry when
TX power > 1.5dBm, attr = &generated1_entryAttr,},};
const RAIL_ChannelConfig_t generated1_channelConfig = { .phyConfigBase = generated1, phyConfigDeltaSubtract = NULL, configs =
generated1_channels, length = 3,};
const uint32_t generated2[] = {...};
RAIL_ChannelConfigEntryAttr_t generated2_entryAttr = {...};
const RAIL_ChannelConfigEntry_t generated2_channels[] = { { .phyConfigDeltaAdd = NULL, baseFrequency = 2450000000, channelSpacing
= 1000000, physicalChannelOffset = 0, channelNumberStart = 0, channelNumberEnd = 20, maxPower = RAIL_TX_POWER_MAX, attr
= &generated2_entryAttr,},};
const RAIL_ChannelConfig_t generated2_channelConfig = { .phyConfigBase = generated2, phyConfigDeltaSubtract = NULL, configs =
generated2_channels, length = 1,};
const RAIL_ChannelConfig_t *channelConfigs[] = { &generated0_channelConfig, &generated1_channelConfig, &generated2_channelConfig,
NULL,}; // ... in main code ... // Create a unique RAIL handle for each unique channel configuration.
railHandle0 = RAIL_Init(&railCfg0, &RAILCb_RfReady0);
railHandle1 = RAIL_Init(&railCfg1, &RAILCb_RfReady1);
railHandle2 = RAIL_Init(&railCfg2, &RAILCb_RfReady2); // Associate each channel configuration with its corresponding RAIL
handle.RAIL_ConfigChannels(railHandle0, channelConfigs[0]); RAIL_ConfigChannels(railHandle1, channelConfigs[1]); RAIL_ConfigChannels(railHandle2,
channelConfigs[2]); // Use a RAIL handle and channel to access the desired channel configuration entry.RAIL_SetTxPowerDbm(railHandle1, 100); //
set 10.0 dBm TX powerRAIL_StartRx(railHandle1, 0, &schedInfo); // RX using generated1_channels[2]RAIL_SetTxPowerDbm(railHandle1, 0); // set 0 dBm
TX powerRAIL_StartRx(railHandle1, 0, &schedInfo); // RX using generated1_channels[1]RAIL_StartRx(railHandle2, 0, &schedInfo); // RX using
generated2_channels[0]

```

Public Attributes

RAIL_RadioConfig_t	phyConfigBase	Base radio configuration for the corresponding channel configuration entries.
RAIL_RadioConfig_t	phyConfigDeltaSubtract	Minimum radio configuration to restore channel entries back to base configuration.
const RAIL_ChannelConfigEntry_t *	configs	Pointer to an array of RAIL_ChannelConfigEntry_t entries.
uint32_t	length	Number of RAIL_ChannelConfigEntry_t entries.
uint32_t	signature	Signature for this structure.

uint32_t [xtalFrequencyHz](#)
Crystal Frequency for the channel config.

Public Attribute Documentation

phyConfigBase

RAIL_RadioConfig_t RAIL_ChannelConfig_t::phyConfigBase

Base radio configuration for the corresponding channel configuration entries.

Definition at line 2356 of file `common/rail_types.h`

phyConfigDeltaSubtract

RAIL_RadioConfig_t RAIL_ChannelConfig_t::phyConfigDeltaSubtract

Minimum radio configuration to restore channel entries back to base configuration.

Definition at line 2361 of file `common/rail_types.h`

configs

const RAIL_ChannelConfigEntry_t* RAIL_ChannelConfig_t::configs

Pointer to an array of [RAIL_ChannelConfigEntry_t](#) entries.

Definition at line 2363 of file `common/rail_types.h`

length

uint32_t RAIL_ChannelConfig_t::length

Number of [RAIL_ChannelConfigEntry_t](#) entries.

Definition at line 2365 of file `common/rail_types.h`

signature

uint32_t RAIL_ChannelConfig_t::signature

Signature for this structure.

Only used on modules.

Definition at line 2367 of file `common/rail_types.h`

xtalFrequencyHz

```
uint32_t RAIL_ChannelConfig_t::xtalFrequencyHz
```

Crystal Frequency for the channel config.

Definition at line 2369 of file common/rail_types.h

RAIL_ChannelMetadata_t

Container for individual channel metadata.

Public Attributes

uint16_t	channel	Channel number.
uint16_t	reserved	Word alignment.
uint32_t	frequency	Channel frequency, in Hz.

Public Attribute Documentation

channel

uint16_t RAIL_ChannelMetadata_t::channel

Channel number.

Definition at line 2378 of file common/rail_types.h

reserved

uint16_t RAIL_ChannelMetadata_t::reserved

Word alignment.

Definition at line 2380 of file common/rail_types.h

frequency

uint32_t RAIL_ChannelMetadata_t::frequency

Channel frequency, in Hz.

Definition at line 2382 of file common/rail_types.h

RAIL_StackInfoCommon_t

Stack info fields common to all protocols.

Public Attributes

uint8_t	protocolId Same as RAIL_PtiProtocol_t .
uint8_t	phyId PHY Id depending on the protocolId value.

Public Attribute Documentation

protocolId

```
uint8_t RAIL_StackInfoCommon_t::protocolId
```

Same as [RAIL_PtiProtocol_t](#).

Definition at line 2391 of file `common/rail_types.h`

phyId

```
uint8_t RAIL_StackInfoCommon_t::phyId
```

PHY Id depending on the protocolId value.

Definition at line 2393 of file `common/rail_types.h`

RAIL_ChannelConfigEntryAttr_t

A channel configuration entry attribute structure.

Items listed are designed to be altered and updated during run-time.

Receive

Receive

APIs related to packet receive.

Modules

[RAIL_ScheduleRxConfig_t](#)

[RAIL_RxPacketInfo_t](#)

[RAIL_RxPacketDetails_t](#)

[RAIL_PrsLnaBypassConfig_t](#)

[Address Filtering](#)

[Packet Information](#)

Enumerations

```
enum RAIL\_RxOptions\_t {
    RAIL_RX_OPTION_STORE_CRC_SHIFT = 0
    RAIL_RX_OPTION_IGNORE_CRC_ERRORS_SHIFT = 1
    RAIL_RX_OPTION_ENABLE_DUALSYNC_SHIFT = 2
    RAIL_RX_OPTION_TRACK_ABORTED_FRAMES_SHIFT = 3
    RAIL_RX_OPTION_REMOVE_APPENDED_INFO_SHIFT = 4
    RAIL_RX_OPTION_ANTENNA0_SHIFT = 5
    RAIL_RX_OPTION_ANTENNA1_SHIFT = 6
    RAIL_RX_OPTION_DISABLE_FRAME_DETECTION_SHIFT = 7
    RAIL_RX_OPTION_CHANNEL_SWITCHING_SHIFT = 10
    RAIL_RX_OPTION_FAST_RX2RX_SHIFT = 11
    RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION_SHIFT = 12
}
```

Receive options, in reality a bitmask.

```
enum RAIL\_RxPacketStatus\_t {
    RAIL_RX_PACKET_NONE = 0
    RAIL_RX_PACKET_ABORT_FORMAT = 1
    RAIL_RX_PACKET_ABORT_FILTERED = 2
    RAIL_RX_PACKET_ABORT_ABORTED = 3
    RAIL_RX_PACKET_ABORT_OVERFLOW = 4
    RAIL_RX_PACKET_ABORT_CRC_ERROR = 5
    RAIL_RX_PACKET_READY_CRC_ERROR = 6
    RAIL_RX_PACKET_READY_SUCCESS = 7
    RAIL_RX_PACKET_RECEIVING = 8
}
```

The packet status code associated with a packet received or currently being received.

Typedefs

```
typedef const void * RAIL\_RxPacketHandle\_t
    A handle used to reference a packet during reception processing.

typedef uint8_t(* RAIL\_ConvertLqiCallback\_t)(uint8_t lqi, int8_t rssi)
    A pointer to a function called before LQI is copied into the RAIL\_RxPacketDetails\_t structure.
```

Functions

RAIL_Status_t	RAIL_ConfigRxOptions (RAIL_Handle_t railHandle, RAIL_RxOptions_t mask, RAIL_RxOptions_t options) Configure receive options.
RAIL_Status_t	RAIL_IncludeFrameTypeLength (RAIL_Handle_t railHandle) Include the code necessary for frame type based length decoding.
void	RAILCb_ConfigFrameTypeLength (RAIL_Handle_t railHandle, const RAIL_FrameType_t *frameType) Handle frame type length.
RAIL_Status_t	RAIL_StartRx (RAIL_Handle_t railHandle, uint16_t channel, const RAIL_SchedulerInfo_t *schedulerInfo) Start the receiver on a specific channel.
RAIL_Status_t	RAIL_ScheduleRx (RAIL_Handle_t railHandle, uint16_t channel, const RAIL_ScheduleRxConfig_t *cfg, const RAIL_SchedulerInfo_t *schedulerInfo) Schedule a receive window for some future time.
RAIL_Status_t	RAIL_EnablePrsLnaBypass (RAIL_Handle_t railHandle, bool enable, const RAIL_PrsLnaBypassConfig_t *pPrsLnaBypassConfig) Enable automatic PRS LNA bypass for external FEM.
RAIL_RxPacketHandle_t	RAIL_HoldRxPacket (RAIL_Handle_t railHandle) Place a temporary hold on this packet's data in the receive FIFO and internal packet metadata FIFO.
uint16_t	RAIL_PeekRxPacket (RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, uint8_t *pDst, uint16_t len, uint16_t offset) Copy 'len' bytes of packet data starting from 'offset' from the receive FIFO.
RAIL_Status_t	RAIL_ReleaseRxPacket (RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle) Release RAIL's resources for a packet previously held in the receive FIFO and internal receive metadata FIFO.
int16_t	RAIL_GetRssi (RAIL_Handle_t railHandle, bool wait) Return the current raw RSSI.
int16_t	RAIL_GetRssiAlt (RAIL_Handle_t railHandle, RAIL_Time_t waitTimeout) Return the current raw RSSI in quarter-dBm within a definitive time period.
RAIL_Status_t	RAIL_StartAverageRssi (RAIL_Handle_t railHandle, uint16_t channel, RAIL_Time_t averagingTimeUs, const RAIL_SchedulerInfo_t *schedulerInfo) Start the RSSI averaging over a specified time in us.
bool	RAIL_IsAverageRssiReady (RAIL_Handle_t railHandle) Query whether the RSSI averaging is done.
int16_t	RAIL_GetAverageRssi (RAIL_Handle_t railHandle) Get the RSSI in quarter-dBm averaged over a specified time in microseconds.
RAIL_Status_t	RAIL_SetRssiOffset (RAIL_Handle_t railHandle, int8_t rssiOffset) Set an RSSI offset.
int8_t	RAIL_GetRssiOffset (RAIL_Handle_t railHandle) Get the radio or protocol RSSI offset in dB.
RAIL_Status_t	RAIL_SetRssiDetectThreshold (RAIL_Handle_t railHandle, int8_t rssiThresholdDbm) Set the RSSI detection threshold in dBm to trigger RAIL_EVENT_DETECT_RSSI_THRESHOLD .
int8_t	RAIL_GetRssiDetectThreshold (RAIL_Handle_t railHandle) Get the RSSI detection threshold in dBm.
int8_t	RAIL_GetRxIncomingPacketRssi (RAIL_Handle_t railHandle) Return the RSSI associated with the incoming packet.

[RAIL_Status_t](#)[RAIL_ConvertLqi](#)(RAIL_Handle_t railHandle, RAIL_ConvertLqiCallback_t cb)

Set up a callback function capable of converting a RX packet's LQI value before being consumed by application code.

Macros

#define	RAIL_RX_OPTIONS_NONE 0	A value representing no options enabled.
#define	RAIL_RX_OPTIONS_DEFAULT RAIL_RX_OPTIONS_NONE	All options are disabled by default.
#define	RAIL_RX_OPTION_STORE_CRC (1UL << RAIL_RX_OPTION_STORE_CRC_SHIFT)	An option to configure whether the CRC portion of the packet is included in the packet payload exposed to the app on packet reception.
#define	RAIL_RX_OPTION_IGNORE_CRC_ERRORS (1UL << RAIL_RX_OPTION_IGNORE_CRC_ERRORS_SHIFT)	An option to configure whether CRC errors will be ignored.
#define	RAIL_RX_OPTION_ENABLE_DUALSYNC (1UL << RAIL_RX_OPTION_ENABLE_DUALSYNC_SHIFT)	An option to control which sync words will be accepted.
#define	RAIL_RX_OPTION_TRACK_ABORTED_FRAMES (1UL << RAIL_RX_OPTION_TRACK_ABORTED_FRAMES_SHIFT)	An option to configure whether frames which are aborted during reception should continue to be tracked.
#define	RAIL_RX_OPTION_REMOVE_APPENDED_INFO (1UL << RAIL_RX_OPTION_REMOVE_APPENDED_INFO_SHIFT)	An option to suppress capturing the appended information after received frames.
#define	RAIL_RX_OPTION_ANTENNA0 (1UL << RAIL_RX_OPTION_ANTENNA0_SHIFT)	An option to select the use of antenna 0 during receive (including Auto-Ack receive).
#define	RAIL_RX_OPTION_ANTENNA1 (1UL << RAIL_RX_OPTION_ANTENNA1_SHIFT)	An option to select the use of antenna 1 during receive (including Auto-Ack receive).
#define	RAIL_RX_OPTION_ANTENNA_AUTO (RAIL_RX_OPTION_ANTENNA0 RAIL_RX_OPTION_ANTENNA1)	An option combination to automatically choose an antenna during receive (including Auto-Ack receive).
#define	RAIL_RX_OPTION_DISABLE_FRAME_DETECTION (1UL << RAIL_RX_OPTION_DISABLE_FRAME_DETECTION_SHIFT)	An option to disable frame detection.
#define	RAIL_RX_OPTION_CHANNEL_SWITCHING (1U << RAIL_RX_OPTION_CHANNEL_SWITCHING_SHIFT)	An option to enable IEEE 802.15.4 RX channel switching.
#define	RAIL_RX_OPTION_FAST_RX2RX (1U << RAIL_RX_OPTION_FAST_RX2RX_SHIFT)	An option to enable fast RX2RX state transition.
#define	RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION (1U << RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION_SHIFT)	An option to enable collision detection.
#define	RAIL_RX_OPTIONS_ALL 0xFFFFFFFFUL	A value representing all possible options.
#define	RAIL_RSSI_INVALID_DBM (-128)	The value returned by RAIL for an invalid RSSI, in dBm.
#define	RAIL_RSSI_INVALID ((int16_t)(RAIL_RSSI_INVALID_DBM * 4))	The value returned by RAIL for an invalid RSSI: in quarter dBm.
#define	RAIL_RSSI_LOWEST ((int16_t)(RAIL_RSSI_INVALID + 1))	The lowest RSSI value returned by RAIL: in quarter dBm.

```
#define RAIL_RSSI_OFFSET_MAX 35
Maximum absolute value for RSSI offset.

#define RAIL_GET_RSSI_WAIT_WITHOUT_TIMEOUT ((RAIL_Time_t)0xFFFFFFFFU)
A sentinel value to indicate waiting for a valid RSSI without a timeout.

#define RAIL_GET_RSSI_NO_WAIT ((RAIL_Time_t)0U)
A sentinel value to indicate no waiting for a valid RSSI.

#define RAIL_RX_PACKET_HANDLE_INVALID (NULL)
An invalid RX packet handle value.

#define RAIL_RX_PACKET_HANDLE_OLDEST ((RAIL_RxPacketHandle_t) 1)
A special RX packet handle to refer to the oldest unreleased packet.

#define RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE ((RAIL_RxPacketHandle_t) 2)
A special RX packet handle to refer to the oldest unreleased complete packet.

#define RAIL_RX_PACKET_HANDLE_NEWEST ((RAIL_RxPacketHandle_t) 3)
A special RX packet handle to refer to the newest unreleased packet when in callback context.
```

Enumeration Documentation

RAIL_RxOptions_t

RAIL_RxOptions_t

Receive options, in reality a bitmask.

Enumerator	
RAIL_RX_OPTION_STORE_CRC_SHIFT	Shift position of RAIL_RX_OPTION_STORE_CRC bit.
RAIL_RX_OPTION_IGNORE_CRC_ERRORS_SHIFT	Shift position of RAIL_RX_OPTION_IGNORE_CRC_ERRORS bit.
RAIL_RX_OPTION_ENABLE_DUALSYNC_SHIFT	Shift position of RAIL_RX_OPTION_ENABLE_DUALSYNC bit.
RAIL_RX_OPTION_TRACK_ABORTED_FRAMES_SHIFT	Shift position of RAIL_RX_OPTION_TRACK_ABORTED_FRAMES bit.
RAIL_RX_OPTION_REMOVE_APPENDED_INFO_SHIFT	Shift position of RAIL_RX_OPTION_REMOVE_APPENDED_INFO bit.
RAIL_RX_OPTION_ANTENNA0_SHIFT	Shift position of RAIL_RX_OPTION_ANTENNA0 bit.
RAIL_RX_OPTION_ANTENNA1_SHIFT	Shift position of RAIL_RX_OPTION_ANTENNA1 bit.
RAIL_RX_OPTION_DISABLE_FRAME_DETECTION_SHIFT	Shift position of RAIL_RX_OPTION_DISABLE_FRAME_DETECTION bit.
RAIL_RX_OPTION_CHANNEL_SWITCHING_SHIFT	Shift position of RAIL_RX_OPTION_CHANNEL_SWITCHING bit.
RAIL_RX_OPTION_FAST_RX2RX_SHIFT	Shift position of RAIL_RX_OPTION_FAST_RX2RX bit.
RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION_SHIFT	Shift position of RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION bit.

Definition at line 3798 of file common/rail_types.h

RAIL_RxPacketStatus_t

RAIL_RxPacketStatus_t

The packet status code associated with a packet received or currently being received.

Note

- RECEIVING implies some packet data may be available, but is untrustworthy (not CRC-verified) and might disappear if the packet is rolled back on error. No packet details are yet available.
- In RX [RAIL_DataMethod_t::FIFO_MODE](#), ABORT statuses imply some packet data may be available, but it's incomplete and not trustworthy.

Enumerator	
RAIL_RX_PACKET_NONE	The radio is idle or searching for a packet.
RAIL_RX_PACKET_ABORT_FORMAT	The packet was aborted during filtering because of illegal frame length, CRC or block decoding errors, other RAIL built-in protocol-specific packet content errors, or by the application or multiprotocol scheduler idling the radio with RAIL_IDLE_ABORT or higher.
RAIL_RX_PACKET_ABORT_FILTERED	The packet failed address filtering.
RAIL_RX_PACKET_ABORT_ABORTED	The packet passed any filtering but was aborted by the application or multiprotocol scheduler idling the radio with RAIL_IDLE_ABORT or higher.
RAIL_RX_PACKET_ABORT_OVERFLOW	The packet overflowed the receive buffer.
RAIL_RX_PACKET_ABORT_CRC_ERROR	The packet passed any filtering but subsequently failed CRC check(s) block decoding, or illegal frame length, and was aborted.
RAIL_RX_PACKET_READY_CRC_ERROR	The packet passed any filtering but subsequently failed CRC check(s) with RAIL_RX_OPTION_IGNORE_CRC_ERRORS in effect.
RAIL_RX_PACKET_READY_SUCCESS	The packet was successfully received, passing CRC check(s).
RAIL_RX_PACKET_RECEIVING	A packet is being received and is not yet complete.

Definition at line 4087 of file `common/rail_types.h`

Typedef Documentation

RAIL_RxPacketHandle_t

RAIL_RxPacketHandle_t

A handle used to reference a packet during reception processing.

There are several sentinel handle values that pertain to certain circumstances: [RAIL_RX_PACKET_HANDLE_INVALID](#), [RAIL_RX_PACKET_HANDLE_OLDEST](#), [RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE](#) and [RAIL_RX_PACKET_HANDLE_NEWEST](#).

Definition at line 4172 of file `common/rail_types.h`

RAIL_ConvertLqiCallback_t

RAIL_ConvertLqiCallback_t)(uint8_t lqi, int8_t rssi)

A pointer to a function called before LQI is copied into the [RAIL_RxPacketDetails_t](#) structure.

Parameters

[in]	lqi	The LQI value obtained by hardware and being readied for application consumption. This LQI value is in integral units ranging from 0 to 255.
[in]	rssi	The RSSI value corresponding to the packet from which the hardware LQI value was obtained. This RSSI value is in integral dBm units.

Returns

- uint8_t The converted LQI value that will be loaded into the [RAIL_RxPacketDetails_t](#) structure in preparation for application consumption. This value should likewise be in integral units ranging from 0 to 255.

Definition at line 4389 of file `common/rail_types.h`

Function Documentation

RAIL_ConfigRxOptions

RAIL_Status_t RAIL_ConfigRxOptions ([RAIL_Handle_t](#) railHandle, [RAIL_RxOptions_t](#) mask, [RAIL_RxOptions_t](#) options)

Configure receive options.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mask	A bitmask containing which options should be modified.
[in]	options	A bitmask containing desired configuration settings. Bit positions for each option are found in the RAIL_RxOptions_t .

Returns

- Status code indicating success of the function call.

Configure the radio receive flow based on the list of available options. Only the options indicated by the mask parameter will be affected. Pass [RAIL_RX_OPTIONS_ALL](#) to set all parameters. The previous settings may affect the current frame if a packet is received during this configuration.

Definition at line 3783 of file `common/rail.h`

RAIL_IncludeFrameTypeLength

RAIL_Status_t RAIL_IncludeFrameTypeLength ([RAIL_Handle_t](#) railHandle)

Include the code necessary for frame type based length decoding.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

This function must be called before [RAIL_ConfigChannels\(\)](#) to allow configurations using a frame type based length setup. It is called by default in the [RAILCb_ConfigFrameTypeLength\(\)](#) API which can be overridden to save code space.

Definition at line 3800 of file `common/rail.h`

RAILCb_ConfigFrameTypeLength

void RAILCb_ConfigFrameTypeLength ([RAIL_Handle_t](#) railHandle, const [RAIL_FrameType_t](#) * frameType)

Handle frame type length.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	frameType	A pointer to a frame type configuration structure.

This function is implemented in the radio configuration. Currently, the frame type passed in only handles packet length decoding. If NULL is passed into this function, it clears any currently configured frame type settings. This will either be implemented as an empty function in the radio configuration if it is not needed, to assist in dead code elimination.

Definition at line 3815 of file common/rail.h

RAIL_StartRx

RAIL_Status_t RAIL_StartRx (RAIL_Handle_t railHandle, uint16_t channel, const RAIL_SchedulerInfo_t * schedulerInfo)

Start the receiver on a specific channel.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	The channel to listen on.
[in]	schedulerInfo	A pointer to information to allow the radio scheduler to place this receive appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call.

This is a non-blocking function. Whenever a packet is received, RAIL_Config_t::eventsCallback will fire with RAIL_EVENT_RX_PACKET_RECEIVED set. If you call this while not idle but with a different channel, any ongoing receive or transmit operation will be aborted.

Definition at line 3835 of file common/rail.h

RAIL_ScheduleRx

RAIL_Status_t RAIL_ScheduleRx (RAIL_Handle_t railHandle, uint16_t channel, const RAIL_ScheduleRxConfig_t * cfg, const RAIL_SchedulerInfo_t * schedulerInfo)

Schedule a receive window for some future time.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	A channel to listen on.
[in]	cfg	A pointer to a configuration structure to define the receive window.
[in]	schedulerInfo	A pointer to information to allow the radio scheduler to place this receive appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call.

This API immediately changes the channel and schedules receive to start at the specified time and end at the given end time. If you do not specify an end time, you may call this API later with an end time as long as you set the start time to disabled. You can also terminate the receive operation immediately using the [RAIL_Idle\(\)](#) function. Note that relative end times are always relative to the start unless no start time is specified. If changing channels, the channel is changed immediately and will abort any ongoing packet transmission or reception.

Returns an error if a CSMA or LBT transmit is still in progress.

In multiprotocol, ensure that you properly yield the radio after this call. See [Yielding the Radio](#) for more details.

Definition at line 3864 of file `common/rail.h`

RAIL_EnablePrsLnaBypass

RAIL_Status_t RAIL_EnablePrsLnaBypass ([RAIL_Handle_t](#) railHandle, bool enable, const [RAIL_PrsLnaBypassConfig_t](#) * pPrsLnaBypassConfig)

Enable automatic PRS LNA bypass for external FEM.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.
[in]	enable	Enable/Disable automatic PRS LNA bypass.
[in]	pPrsLnaBypassConfig	A pointer to an automatic PRS LNA bypass configuration structure. It must be non-NULL to enable the feature.

Returns

- Status code indicating success of the function call.

If automatic PRS LNA bypass is enabled on chip that supports the feature ([RAIL_SUPPORTS_PRS_LNA_BYPASS](#)), a level is generated on a PRS channel when the received power exceed a threshold. It is turned off after frame reception or after timeout if no frame has been detected.

Warnings

- Calling this function before [RAIL_Init\(\)](#) won't enable the the feature now. It will return [RAIL_STATUS_NO_ERROR](#) but the feature will be enabled by [RAIL_Init\(\)](#).
- As this function relies on PRS access and RAIL is meant to run in TrustZone non-secure world, it is not supported if PRS is configured as secure peripheral and it will return [RAIL_STATUS_INVALID_CALL](#).

Definition at line 3891 of file `common/rail.h`

RAIL_HoldRxPacket

RAIL_RxPacketHandle_t RAIL_HoldRxPacket ([RAIL_Handle_t](#) railHandle)

Place a temporary hold on this packet's data in the receive FIFO and internal packet metadata FIFO.

Parameters

[in]	railHandle	A RAIL instance handle.

This function can only be called from within RAIL callback context. It can be used in any RX mode.

Normally, when RAIL issues its callback indicating a packet is ready or aborted, it expects the application's callback to retrieve and copy (or discard) the packet's information and data, and will free the packet data and metadata after the callback returns. This function tells RAIL to hold onto those resources in the receive FIFO and internal metadata FIFO after

the callback returns in case the application wants to defer processing the packet to a later time, e.g., outside of callback context.

Returns

- The packet handle for the packet associated with the callback, [RAIL_RX_PACKET_HANDLE_INVALID](#) if no such packet yet exists or railHandle is not active.

Note

- When using multiprotocol the receive FIFO and internal receive metadata FIFO are reset during protocol switches losing any held packets. It is best to avoid using [RAIL_HoldRxPacket](#) in DMP or to at least invalidate any held packet handles and related [RAIL_RxPacketInfo_t](#) information when the [RAIL_EVENT_CONFIG_UNSCHEDULED](#) occurs.

Definition at line 4279 of file `common/rail.h`

RAIL_PeekRxPacket

```
uint16_t RAIL_PeekRxPacket (RAIL\_Handle\_t railHandle, RAIL\_RxPacketHandle\_t packetHandle, uint8_t * pDst, uint16_t len,
uint16_t offset)
```

Copy 'len' bytes of packet data starting from 'offset' from the receive FIFO.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	packetHandle	A packet handle as returned from a previous RAIL_GetRxPacketInfo() or RAIL_HoldRxPacket() call, or sentinel values RAIL_RX_PACKET_HANDLE_OLDEST , RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE or RAIL_RX_PACKET_HANDLE_NEWEST .
[out]	pDst	A pointer to the location where the received bytes will be copied. If NULL, no copying occurs.
[in]	len	A number of packet data bytes to copy.
[in]	offset	A byte offset within remaining packet data from which to copy.

Those bytes remain valid for re-peeking.

Returns

- Number of packet bytes copied.

Note

- Peek does not permit peeking beyond the requested packet's available packet data (though there is a small chance it might for a [RAIL_RX_PACKET_HANDLE_NEWEST](#) packet at the very end of still being received). Nor can one peek into already-consumed data read by [RAIL_ReadRxFifo\(\)](#). len and offset are relative to the remaining data available in the packet, if any was already consumed by [RAIL_ReadRxFifo\(\)](#).

Definition at line 4305 of file `common/rail.h`

RAIL_ReleaseRxPacket

```
RAIL_Status_t RAIL_ReleaseRxPacket (RAIL\_Handle\_t railHandle, RAIL\_RxPacketHandle\_t packetHandle)
```

Release RAIL's resources for a packet previously held in the receive FIFO and internal receive metadata FIFO.

Parameters

[in]	railHandle	A RAIL instance handle.

[in]	packetHandle	A packet handle as returned from a previous RAIL_HoldRxPacket() call, or sentinel values RAIL_RX_PACKET_HANDLE_OLDEST , RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE or RAIL_RX_PACKET_HANDLE_NEWEST . The latter might be used within RAIL callback context to explicitly release the packet associated with the callback early, before it's released automatically by RAIL on callback return (unless explicitly held).

This function must be called for any packet previously held via [RAIL_HoldRxPacket\(\)](#). It may optionally be called within a callback context to release RAIL resources sooner than at callback completion time when not holding the packet. This function can be used in any RX mode.

Returns

- [RAIL_STATUS_NO_ERROR](#) if the held packet was released or an appropriate error code otherwise.

Definition at line 4334 of file common/rail.h

RAIL_GetRssi

```
int16_t RAIL_GetRssi (RAIL_Handle_t railHandle, bool wait)
```

Return the current raw RSSI.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	wait	if false returns instant RSSI with no checks.

Returns

- [RAIL_RSSI_INVALID](#) if the receiver is disabled and an RSSI value can't be obtained. Otherwise, return the RSSI in quarter dBm, dbm*4.

Gets the current RSSI value. This value represents the current energy of the channel. It can change rapidly and will be low if no RF energy is in the current channel. The function from the value reported to dBm is an offset dependent on the PHY and the PCB layout. Characterize the RSSI received on your hardware and apply an offset in the application to account for board and PHY parameters. When 'wait' is false, the radio needs to be currently in RX and have been in there for a sufficient amount of time for a fresh RSSI value to be read and returned. Otherwise, the RSSI is considered stale and [RAIL_RSSI_INVALID](#) is returned instead. When 'wait' is true, if the radio is transitioning to or already in RX, this function will wait for a valid RSSI to be read and return it. Otherwise, if the radio is in or transitions to IDLE or TX, [RAIL_RSSI_INVALID](#) will be returned. On low datarate PHYs, this function can take a significantly longer time when wait is true.

In multiprotocol, this function returns [RAIL_RSSI_INVALID](#) immediately if railHandle is not the currently active [RAIL_Handle_t](#). Additionally, 'wait' should never be set 'true' in multiprotocol as the wait time is not consistent, so scheduling a scheduler slot cannot be done accurately. Rather if waiting for a valid RSSI is desired, use [RAIL_GetRssiAlt](#) instead to apply a bounded time period.

Note

- If RX Antenna Diversity is enabled via [RAIL_ConfigRxOptions\(\)](#), pass true for the wait parameter otherwise it's very likely [RAIL_RSSI_INVALID](#) will be returned.
- If RX channel hopping is turned on, do not use this API. Instead, see [RAIL_GetChannelHoppingRssi\(\)](#).
- When 'wait' is false, this API is equivalent to [RAIL_GetRssiAlt](#) with 'waitTimeout' set to [RAIL_GET_RSSI_NO_WAIT](#). When 'wait' is true, this API is equivalent to [RAIL_GetRssiAlt](#) with 'waitTimeout' set to [RAIL_GET_RSSI_WAIT_WITHOUT_TIMEOUT](#). Consider using [RAIL_GetRssiAlt](#) if a bounded maximum wait timeout is desired.

Definition at line 4380 of file common/rail.h

RAIL_GetRssiAlt


```
int16_t RAIL_GetRssiAlt (RAIL_Handle_t railHandle, RAIL_Time_t waitTimeout)
```

Return the current raw RSSI in quarter-dBm within a definitive time period.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	waitTimeout	Sets the maximum time in microseconds to wait for a valid RSSI. If equal to RAIL_GET_RSSI_NO_WAIT returns instant RSSI with no checks. If equal to RAIL_GET_RSSI_WAIT_WITHOUT_TIMEOUT waits for a valid RSSI with no maximum timeout.

Returns

- [RAIL_RSSI_INVALID](#) if the receiver is disabled and an RSSI value can't be obtained. Otherwise, return the RSSI in quarter dBm (dBm*4).

Gets the current RSSI value. This value represents the current energy of the channel. It can change rapidly, and will be low if no RF energy is in the current channel. The function from the value reported to dBm is an offset dependent on the PHY and the PCB layout. Characterize the RSSI received on your hardware and apply an offset in the application to account for board and PHY parameters. If a value of [RAIL_GET_RSSI_NO_WAIT](#) is given for waitTimeout, the radio needs to be currently in RX and have been in there for a sufficient amount of time for a fresh RSSI value to be read and returned. Otherwise the RSSI is considered stale and [RAIL_RSSI_INVALID](#) is returned instead. For non-zero values of waitTimeout, if the radio is transitioning to or already in RX, this function will wait a maximum time equal to waitTimeout (or indefinitely if waitTimeout is set to [RAIL_GET_RSSI_WAIT_WITHOUT_TIMEOUT](#)) for a valid RSSI to be read and return it. Otherwise, if the waitTimeout is reached, or the radio is in or transitions to IDLE or TX, [RAIL_RSSI_INVALID](#) will be returned. On low datarate PHYs, this function can take a significantly longer time when waitTimeout is non-zero.

In multiprotocol, this function returns [RAIL_RSSI_INVALID](#) immediately if railHandle is not the currently active [RAIL_Handle_t](#). Additionally, waitTimeout should never be set to a value other than [RAIL_GET_RSSI_NO_WAIT](#) in multiprotocol as the integration between this feature and the radio scheduler has not been implemented.

Note

- If RX Antenna Diversity is enabled via [RAIL_ConfigRxOptions\(\)](#), pass true for the wait parameter otherwise it's very likely [RAIL_RSSI_INVALID](#) will be returned.
- If RX Antenna Diversity is enabled via [RAIL_ConfigRxOptions\(\)](#), the RSSI value returned could come from either antenna and vary between antennas.
- If RX channel hopping is turned on, do not use this API. Instead, see [RAIL_GetChannelHoppingRssi\(\)](#).

Definition at line 4426 of file common/rail.h

RAIL_StartAverageRssi

```
RAIL_Status_t RAIL_StartAverageRssi (RAIL_Handle_t railHandle, uint16_t channel, RAIL_Time_t averagingTimeUs, const RAIL_SchedulerInfo_t * schedulerInfo)
```

Start the RSSI averaging over a specified time in us.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	The physical channel to set.
[in]	averagingTimeUs	Averaging time in microseconds.
[in]	schedulerInfo	A pointer to information to allow the radio scheduler to place this operation appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

Status code indicating success of the function call.

Starts a non-blocking hardware-based RSSI averaging mechanism. Only a single instance of RSSI averaging can be run at any time and the radio must be idle to start.

In multiprotocol, this is a scheduled event. It will start when railHandle becomes active. railHandle needs to stay active until the averaging completes. If the averaging is interrupted, calls to `RAIL_GetAverageRssi()` will return `RAIL_RSSI_INVALID`.

Also in multiprotocol, the user is required to call `RAIL_YieldRadio()` after this event completes (i.e., when `RAIL_EVENT_RSSI_AVERAGE_DONE` occurs).

Note

- If the radio is idled while RSSI averaging is still in effect, a `RAIL_EVENT_RSSI_AVERAGE_DONE` event may not occur and `RAIL_IsAverageRssiReady()` may never return true.
- Completion of RSSI averaging, marked by RAIL event `RAIL_EVENT_RSSI_AVERAGE_DONE`, will return the radio to idle state.

Definition at line 4459 of file common/rail.h

RAIL_IsAverageRssiReady

```
bool RAIL_IsAverageRssiReady (RAIL_Handle_t railHandle)
```

Query whether the RSSI averaging is done.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if done and false otherwise.

This function can be used to poll for completion of the RSSI averaging to avoid relying on an interrupt-based callback.

Note

- If the radio is idled while RSSI averaging is still in effect, this function may never return true.

Definition at line 4476 of file common/rail.h

RAIL_GetAverageRssi

```
int16_t RAIL_GetAverageRssi (RAIL_Handle_t railHandle)
```

Get the RSSI in quarter-dBm averaged over a specified time in microseconds.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The RSSI in quarter-dBm (dBm * 4), or `RAIL_RSSI_INVALID` if the receiver is disabled or an an RSSI value couldn't be obtained.

Gets the hardware RSSI average after issuing `RAIL_StartAverageRssi()`. Use after `RAIL_StartAverageRssi()`.

Definition at line 4488 of file common/rail.h

RAIL_SetRssiOffset

```
RAIL_Status_t RAIL_SetRssiOffset (RAIL_Handle_t railHandle, int8_t rssiOffset)
```

Set an RSSI offset.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.
[in]	rssiOffset	desired dB offset to be added to the RSSI measurements.

Returns

- Status code indicating success of the function call. [RAIL_STATUS_INVALID_CALL](#) if called with radio-generic handle, such as [RAIL_EFR32_HANDLE](#), after RAIL initialization. [RAIL_STATUS_INVALID_PARAMETER](#) if the RSSI offset is deemed large enough to cause the RSSI readings to underflow or overflow.

Adds an offset to the RSSI in dBm. This offset affects all functionality that depends on RSSI values, such as CCA functions. Do not modify the offset dynamically during packet reception. This function can only be called while the radio is off, or in the case of multiprotocol, on an inactive protocol.

Note

- : Prior to RAIL being initialized, a radio-wide protocol-independent offset can be established using a radio-generic handle like [RAIL_EFR32_HANDLE](#). The radio RSSI offset can range from -50 to +20 dB. After RAIL has been initialized a real RAIL protocol handle must be provided to set a protocol-specific RSSI offset, which is added to any radio offset (plus a per-PHY offset set by the radio calculator). The absolute value of the overall sum of all these offsets cannot exceed [RAIL_RSSI_OFFSET_MAX](#) otherwise RSSIs could underflow or overflow.
- : During [RX Channel Hopping](#) this API will not update the RSSI offset immediately if channel hopping has already been configured. A subsequent call to [RAIL_ZWAVE_ConfigRxChannelHopping\(\)](#) or [RAIL_ConfigRxChannelHopping\(\)](#) is required for the new RSSI offset to take effect.

Definition at line 4522 of file common/rail.h

RAIL_GetRssiOffset

```
int8_t RAIL_GetRssiOffset (RAIL_Handle_t railHandle)
```

Get the radio or protocol RSSI offset in dB.

Parameters

[in]	railHandle	A radio-generic or real RAIL instance handle.

Returns

- The RSSI offset in dB corresponding to the railHandle.

Note

- : A radio-generic handle, such as [RAIL_EFR32_HANDLE](#), can be used to get the radio's RSSI offset otherwise this will return the RSSI offset value associated with the RAIL protocol instance handle, exclusive of any radio or PHY-specific RSSI offset correction, if any.

Definition at line 4535 of file common/rail.h

RAIL_SetRssiDetectThreshold

RAIL_Status_t RAIL_SetRssiDetectThreshold (RAIL_Handle_t railHandle, int8_t rssiThresholdDbm)

Set the RSSI detection threshold in dBm to trigger [RAIL_EVENT_DETECT_RSSI_THRESHOLD](#).

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	rssiThresholdDbm	desired RSSI threshold in dBm over which the event RAIL_EVENT_DETECT_RSSI_THRESHOLD is triggered.

Returns

- Status code indicating success of the function call. Returns [RAIL_STATUS_INVALID_STATE](#) in multiprotocol, if the requested railHandle is not active. Returns [RAIL_STATUS_INVALID_CALL](#) if called on parts on which this function is not supported.

When in receive, RSSI is sampled and if it exceeds the threshold, [RAIL_EVENT_DETECT_RSSI_THRESHOLD](#) is triggered.

Note

- : If the radio is idled or this function is called with rssiThresholdDbm as [RAIL_RSSI_INVALID_DBM](#) while RSSI detect is still in effect, a [RAIL_EVENT_DETECT_RSSI_THRESHOLD](#) may not occur and the detection is disabled. If the RSSI is already above threshold when this function is called then [RAIL_EVENT_DETECT_RSSI_THRESHOLD](#) will occur. Once the RSSI goes over the configured threshold and [RAIL_EVENT_DETECT_RSSI_THRESHOLD](#) occurs, this function needs to be called again to reactivate the RSSI threshold detection.

This function is only available on EFR32 Series 2 Sub-GHz parts EFR32xG23 and EFR32xG25.

Definition at line 4564 of file common/rail.h

RAIL_GetRssiDetectThreshold

int8_t RAIL_GetRssiDetectThreshold (RAIL_Handle_t railHandle)

Get the RSSI detection threshold in dBm.

Parameters

[in]	railHandle	a RAIL instance handle.

Returns

- The RSSI threshold in dBm corresponding to the railHandle.

Note

- : The function returns [RAIL_RSSI_INVALID_DBM](#) when [RAIL_SetRssiDetectThreshold\(\)](#) is not supported or disabled. In multiprotocol, the function returns [RAIL_RSSI_INVALID_DBM](#) if railHandle is not active.

This function is only available on EFR32 Series 2 Sub-GHz parts EFR32xG23 and EFR32xG25.

Definition at line 4580 of file common/rail.h

RAIL_GetRxIncomingPacketRssi

int8_t RAIL_GetRxIncomingPacketRssi (RAIL_Handle_t railHandle)

Return the RSSI associated with the incoming packet.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The RSSI on the incoming packet in dBm, or [RAIL_RSSI_INVALID_DBM](#) if not available.

This function can only be called from callback context, e.g., when handling [RAIL_EVENT_RX_FILTER_PASSED](#) or [RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#).

Definition at line 4593 of file `common/rail.h`

RAIL_ConvertLqi

RAIL_Status_t RAIL_ConvertLqi ([RAIL_Handle_t](#) railHandle, [RAIL_ConvertLqiCallback_t](#) cb)

Set up a callback function capable of converting a RX packet's LQI value before being consumed by application code.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	cb	A callback of type RAIL_ConvertLqiCallback_t that is called before the RX packet LQI value is loaded into the RAIL_RxPacketDetails_t structure for application consumption.

Returns

- Status code indicating success of the function call.

Definition at line 4605 of file `common/rail.h`

RAIL_ScheduleRxConfig_t

Configures the scheduled RX algorithm.

Defines the start and end times of the receive window created for a scheduled receive. If either start or end times are disabled, they will be ignored.

Public Attributes

RAIL_Time_t	start	The time to start receive.
RAIL_TimeMode_t	startMode	How to interpret the time value specified in the start parameter.
RAIL_Time_t	end	The time to end receive.
RAIL_TimeMode_t	endMode	How to interpret the time value specified in the end parameter.
uint8_t	rxTransitionEndSchedule	While in scheduled RX, you can still control the radio state via state transitions.
uint8_t	hardWindowEnd	This setting tells RAIL what to do with a packet being received when the window end event occurs.

Public Attribute Documentation

start

RAIL_Time_t RAIL_ScheduleRxConfig_t::start

The time to start receive.

See [startMode](#) for more information about the types of start times that you can specify.

Definition at line [4021](#) of file [common/rail_types.h](#)

startMode

RAIL_TimeMode_t RAIL_ScheduleRxConfig_t::startMode

How to interpret the time value specified in the start parameter.

See the [RAIL_TimeMode_t](#) documentation for more information. Use [RAIL_TIME_ABSOLUTE](#) for absolute times, [RAIL_TIME_DELAY](#) for times relative to the current time and [RAIL_TIME_DISABLED](#) to ignore the start time.

Definition at line [4029](#) of file [common/rail_types.h](#)

end

```
RAIL_Time_t RAIL_ScheduleRxConfig_t::end
```

The time to end receive.

See `endMode` for more information about the types of end times you can specify.

Definition at line 4034 of file `common/rail_types.h`

endMode

```
RAIL_TimeMode_t RAIL_ScheduleRxConfig_t::endMode
```

How to interpret the time value specified in the end parameter.

See the [RAIL_TimeMode_t](#) documentation for more information. Note that, in this API, if you specify a [RAIL_TIME_DELAY](#), it is relative to the start time if given and relative to now if none is specified. Also, using [RAIL_TIME_DISABLED](#) means that this window will not end unless you explicitly call [RAIL_Idle\(\)](#) or add an end event through a future update to this configuration.

Definition at line 4044 of file `common/rail_types.h`

rxTransitionEndSchedule

```
uint8_t RAIL_ScheduleRxConfig_t::rxTransitionEndSchedule
```

While in scheduled RX, you can still control the radio state via state transitions.

This option configures whether a transition to RX goes back to scheduled RX or to the normal RX state. Once in the normal RX state, you will effectively end the scheduled RX window and can continue to receive indefinitely depending on the state transitions. Set to 1 to transition to normal RX and 0 to stay in the scheduled RX.

This setting also influences the posting of [RAIL_EVENT_RX_SCHEDULED_RX_END](#) when the scheduled Rx window is implicitly ended by a packet receive (any of the [RAIL_EVENTS_RX_COMPLETION](#) events). See that event for details.

Note

- An Rx transition to Idle state will always terminate the scheduled Rx window, regardless of this setting. This can be used to ensure Scheduled RX terminates on the first packet received (or first successful packet if the RX error transition is to Rx while the Rx success transition is to Idle).

Definition at line 4064 of file `common/rail_types.h`

hardWindowEnd

```
uint8_t RAIL_ScheduleRxConfig_t::hardWindowEnd
```

This setting tells RAIL what to do with a packet being received when the window end event occurs.

If set to 0, such a packet will be allowed to complete. Any other setting will cause that packet to be aborted. In either situation, any posting of [RAIL_EVENT_RX_SCHEDULED_RX_END](#) is deferred briefly to when the packet's corresponding [RAIL_EVENTS_RX_COMPLETION](#) occurs.

Definition at line 4073 of file common/rail_types.h

RAIL_RxPacketInfo_t

Basic information about a packet being received or already completed and awaiting processing, including memory pointers to its data in the circular receive FIFO buffer.

This packet information refers to remaining packet data that has not already been consumed by [RAIL_ReadRxFifo\(\)](#).

Note

- Because the receive FIFO buffer is circular, a packet might start near the end of the buffer and wrap around to the beginning of the buffer to finish, hence the distinction between the first and last portions. Packets that fit without wrapping only have a first portion (firstPortionBytes == packetBytes and lastPortionData will be NULL).

Public Attributes

RAIL_RxPacketStatus_t	packetStatus The packet status of this packet.
uint16_t	packetBytes The number of packet data bytes available to read in this packet.
uint16_t	firstPortionBytes The number of bytes in the first portion.
uint8_t *	firstPortionData The pointer to the first portion of packet data containing firstPortionBytes number of bytes.
uint8_t *	lastPortionData The pointer to the last portion of a packet, if any; NULL otherwise.
RAIL_AddrFilterMask_t	filterMask A bitmask representing which address filter(s) this packet has passed.

Public Attribute Documentation

packetStatus

RAIL_RxPacketStatus_t RAIL_RxPacketInfo_t::packetStatus

The packet status of this packet.

Definition at line 4218 of file common/rail_types.h

packetBytes

uint16_t RAIL_RxPacketInfo_t::packetBytes

The number of packet data bytes available to read in this packet.

Definition at line 4220 of file common/rail_types.h

firstPortionBytes

```
uint16_t RAIL_RxPacketInfo_t::firstPortionBytes
```

The number of bytes in the first portion.

Definition at line 4222 of file `common/rail_types.h`

firstPortionData

```
uint8_t* RAIL_RxPacketInfo_t::firstPortionData
```

The pointer to the first portion of packet data containing firstPortionBytes number of bytes.

Definition at line 4227 of file `common/rail_types.h`

lastPortionData

```
uint8_t* RAIL_RxPacketInfo_t::lastPortionData
```

The pointer to the last portion of a packet, if any; NULL otherwise.

The number of bytes in this portion is packetBytes - firstPortionBytes.

Definition at line 4233 of file `common/rail_types.h`

filterMask

```
RAIL_AddrFilterMask_t RAIL_RxPacketInfo_t::filterMask
```

A bitmask representing which address filter(s) this packet has passed.

Will be 0 when not filtering or if packet info is retrieved before filtering has completed. It's undefined on platforms lacking [RAIL_SUPPORTS_ADDR_FILTER_MASK](#).

Definition at line 4240 of file `common/rail_types.h`

RAIL_RxPacketDetails_t

Received packet details obtained via [RAIL_GetRxPacketDetails\(\)](#) or [RAIL_GetRxPacketDetailsAlt\(\)](#).

Note

- Certain details are always available, while others are only available if the [RAIL_RxOptions_t](#)[RAIL_RX_OPTION_REMOVE_APPENDED_INFO](#) option is not in effect and the received packet's [RAIL_RxPacketStatus_t](#) is among the [RAIL_RX_PACKET_READY_](#) set. Each detail's description indicates its availability.

Public Attributes

RAIL_PacketTimeStamp_t	timeReceived The timestamp of the received packet in the RAIL timebase.
bool	crcPassed Indicates whether the CRC passed or failed for the received packet.
bool	isAck Indicate whether the received packet was the expected Ack.
int8_t	rssi RSSI of the received packet in integer dBm.
uint8_t	lqi The link quality indicator of the received packet.
uint8_t	syncWordId For radios and PHY configurations that support multiple sync words, this number is the ID of the sync word that was used for this packet.
uint8_t	subPhyId In configurations where the radio has the option of receiving a given packet in multiple ways, indicates which of the sub-PHY options was used to receive the packet.
uint8_t	antennald For Antenna Control configurations where the device has multiple antennas, this indicates which antenna received the packet.
uint8_t	channelHoppingChannelIndex When channel hopping is enabled, this field will contain the index of the channel in RAIL_RxChannelHoppingConfig_t::entries on which this packet was received, or a sentinel value.
uint16_t	channel The channel on which the packet was received.

Public Attribute Documentation

timeReceived

[RAIL_PacketTimeStamp_t](#) [RAIL_RxPacketDetails_t::timeReceived](#)

The timestamp of the received packet in the RAIL timebase.

When not available it will be [RAIL_PACKET_TIME_INVALID](#).

Definition at line 4261 of file common/rail_types.h

crcPassed

```
bool RAIL_RxPacketDetails_t::crcPassed
```

Indicates whether the CRC passed or failed for the received packet.

It is true for [RAIL_RX_PACKET_READY_SUCCESS](#) packets and false for all others.

It is always available.

Definition at line 4269 of file common/rail_types.h

isAck

```
bool RAIL_RxPacketDetails_t::isAck
```

Indicate whether the received packet was the expected Ack.

It is true for the expected Ack and false otherwise.

It is always available.

An expected Ack is defined as a protocol-correct Ack packet successfully-received ([RAIL_RX_PACKET_READY_SUCCESS](#) or [RAIL_RX_PACKET_READY_CRC_ERROR](#)) and whose sync word was detected within the [RAIL_AutoAckConfig_t::ackTimeout](#) period following a transmit which specified [RAIL_TX_OPTION_WAIT_FOR_ACK](#), requested an Ack, and Auto-Ack is enabled. When true, the ackTimeout period was terminated so no [RAIL_EVENT_RX_ACK_TIMEOUT](#) will be subsequently posted for the transmit.

A "protocol-correct Ack" applies to the 802.15.4 or Z-Wave protocols for which RAIL can discern the frame type and match the Ack's sequence number with that of the transmitted frame. For other protocols, the first packet successfully-received whose sync word was detected within the [RAIL_AutoAckConfig_t::ackTimeout](#) period is considered the expected Ack; upper layers are responsible for confirming this.

Definition at line 4295 of file common/rail_types.h

rssI

```
int8_t RAIL_RxPacketDetails_t::rssI
```

RSSI of the received packet in integer dBm.

This RSSI measurement is started as soon as the sync word is detected. The duration of the measurement is PHY-specific.

When not available it will be [RAIL_RSSI_INVALID_DBM](#).

Definition at line 4303 of file common/rail_types.h

lqi

```
uint8_t RAIL_RxPacketDetails_t::lqi
```

The link quality indicator of the received packet.

A zero would indicate a very low quality packet while a 255 would indicate a very high quality packet.

When not available it will be 0.

Definition at line 4311 of file common/rail_types.h

syncWordId

```
uint8_t RAIL_RxPacketDetails_t::syncWordId
```

For radios and PHY configurations that support multiple sync words, this number is the ID of the sync word that was used for this packet.

It is always available.

Definition at line 4318 of file common/rail_types.h

subPhyId

```
uint8_t RAIL_RxPacketDetails_t::subPhyId
```

In configurations where the radio has the option of receiving a given packet in multiple ways, indicates which of the sub-PHY options was used to receive the packet.

Most radio configurations do not have this ability and the subPhyId is set to 0.

Currently, this field is used by the BLE Coded PHY, the BLE Simulscan PHY and the SUN OFDM PHYs. In BLE cases, a value of 0 marks a 500 kbps packet, a value of 1 marks a 125 kbps packet, and a value of 2 marks a 1 Mbps packet. Also, see [RAIL_BLE_ConfigPhyCoded\(\)](#) and [RAIL_BLE_ConfigPhySimulscan\(\)](#).

In SUN OFDM cases, the value corresponds to the numerical value of the Modulation and Coding Scheme (MCS) level of the last received packet. The packet bitrate depends on the MCS value, as well as the OFDM option. Packets bitrates for SUN OFDM PHYs can be found in 802.15.4-2020 specification, chapter 20.3, table 20-10. Ex: Packet bitrate for OFDM option 1 MCS0 is 100kb/s and 2400kb/s for MCS6.

In WMBUS cases, when using PHY_wMbus_ModeTC_M2O_100k_frameA with simultaneous RX of T and C modes enabled ([RAIL_WMBUS_Config\(\)](#)), the value corresponds to [RAIL_WMBUS_Phy_t](#).

It is always available.

Definition at line 4344 of file common/rail_types.h

antennald

```
uint8_t RAIL_RxPacketDetails_t::antennald
```

For [Antenna Control](#) configurations where the device has multiple antennas, this indicates which antenna received the packet.

When there is only one antenna, this will be set to the default of 0.

It is always available.

Definition at line 4352 of file common/rail_types.h

channelHoppingChannelIndex

```
uint8_t RAIL_RxPacketDetails_t::channelHoppingChannelIndex
```

When channel hopping is enabled, this field will contain the index of the channel in [RAIL_RxChannelHoppingConfig_t::entries](#) on which this packet was received, or a sentinel value.

It is always available.

Definition at line 4360 of file common/rail_types.h

channel

```
uint16_t RAIL_RxPacketDetails_t::channel
```

The channel on which the packet was received.

It is always available.

Note

- It is best to fully process (empty or clear) the receive FIFO before changing channel configurations ([RAIL_ConfigChannels\(\)](#) or a built-in configuration) as unprocessed packets' channel could reflect the wrong configuration.

Definition at line 4371 of file common/rail_types.h

RAIL_PrsLnaBypassConfig_t

Configures the automatic PRS LNA bypass.

Public Attributes

- uint32_t

timeoutUs

Maximum time in microseconds to wait for frame detection after the LNA has been bypassed.
- uint8_t

threshold

Threshold (without unit) from which LNA bypass is turned on.
- uint8_t

deltaRssiDbm

Compensation in dBm applied by RAIL to RSSI during LNA bypass.
- uint8_t

prsChannel

PRS Channel used for the bypass.
- bool

polarity

PRS signal polarity for bypass.

Public Attribute Documentation

timeoutUs

uint32_t RAIL_PrsLnaBypassConfig_t::timeoutUs

Maximum time in microseconds to wait for frame detection after the LNA has been bypassed.

It must be greater than 0 to enable automatic PRS LNA bypass with [RAIL_EnablePrsLnaBypass\(\)](#).

Definition at line 4402 of file common/rail_types.h

threshold

uint8_t RAIL_PrsLnaBypassConfig_t::threshold

Threshold (without unit) from which LNA bypass is turned on.

The table below shows EFR32XG25 thresholds corresponding to received power level without the LNA gain.

Level dB	FSK_1a	FSK_1b	FSK_2a	FSK_2b	FSK_3	FSK_4a	FSK_4b	FSK_5	OFDM1	OFDM2	OFDM3	OFDM4
-25									9	9	9	10
-20		7	7	7	8	8	7	8	11	12	12	12
-15	7	10	10	10	9	9	10	10	14	14	14	15
-10	9	12	12	12	12	12	12	12	16	16	16	16
-5	11	14	14	14	16	16	14	16				
0	14	17	18	17	17	18	18	18				

For example, with OFDM1 PHY, setting the threshold to 11 will turn on the bypass when the power level at EFR32XG25 input is greater than -20 dB.

Definition at line 4420 of file common/rail_types.h

deltaRssiDbm

```
uint8_t RAIL_PrsLnaBypassConfig_t::deltaRssiDbm
```

Compensation in dBm applied by RAIL to RSSI during LNA bypass.

The RSSI offset set using [RAIL_SetRssiOffset\(\)](#) must correspond to the case with FEM LNA not bypassed. deltaRssiDbm is typically the FEM LNA gain value.

Definition at line 4427 of file common/rail_types.h

prsChannel

```
uint8_t RAIL_PrsLnaBypassConfig_t::prsChannel
```

PRS Channel used for the bypass.

PRS_GetFreeChannel() can be use to find a free channel. Then the signal can be routed to GPIO pin and port using PRS_PinOutput(). This allows logical operations with other PRS channels and so to adapt to the FEM control logic table. Any call to PRS_Combine() with [RAIL_PrsLnaBypassConfig_t::prsChannel](#) as chA must be done after the [RAIL_EnablePrsLnaBypass\(\)](#) call.

Definition at line 4437 of file common/rail_types.h

polarity

```
bool RAIL_PrsLnaBypassConfig_t::polarity
```

PRS signal polarity for bypass.

With a polarity of 1, PRS signal is set to 1 for bypass and 0 for un-bypass. with a polarity of 0, PRS signal is set to 0 for bypass and 1 for un-bypass.

Definition at line 4444 of file common/rail_types.h

Address Filtering

Address Filtering

Configuration APIs for receive packet address filtering.

The address filtering code examines the packet as follows.

Bytes: 0 - 255	0 - 8	0 - 255	0 - 8	Variable
Data_0	Field_0	Data_1	Field_1	Data_2

In the above structure, anything listed as Data_# is an optional section of bytes that RAIL will not process for address filtering. The Field_# segments reference specific sections in the packet that will each be interpreted as an address during address filtering. The application may submit up to four addresses to attempt to match each field segment and each address may have a size of up to 8 bytes. To set up address filtering, first configure the locations and length of the addresses in the packet. Next, configure which combinations of matches in Field_0 and Field_1 should constitute an address match. Last, enter addresses into tables for each field and enable them. The first two of these are part of the [RAIL_AddrConfig_t](#) structure while the second part is configured at runtime using the [RAIL_SetAddressFilterAddress\(\)](#) API. A brief description of each configuration is listed below.

The offsets and sizes of the fields are assumed fixed for the RAIL address filter. To set them, specify arrays for these values in the sizes and offsets entries in the [RAIL_AddrConfig_t](#) structure. A size of zero indicates that a field is disabled. The start offset for a field is relative to the previous start offset and, if you're using frame type decoding, the first start offset is relative to the end of the byte containing the frame type.

Configuring which combinations of Field_0 and Field_1 constitute a match is the most complex portion of the address filter. The easiest way to think about this is with a truth table. If you consider each of the four possible address entries in a field, you can have a match on any one of those or a match for none of them. This is shown in the 5x5 truth table below where Field_0 matches are the rows and Field_1 matches are the columns.

	No Match	Address 0	Address 1	Address 2	Address 3
No Match	bit 0	bit 1	bit 2	bit 3	bit 4
Address 0	bit 5	bit 6	bit 7	bit 8	bit 9
Address 1	bit 10	bit 11	bit 12	bit 13	bit 14
Address 2	bit 15	bit 16	bit 17	bit 18	bit 19
Address 3	bit 20	bit 21	bit 22	bit 23	bit 24

Because this is only 25 bits, it can be represented in one 32-bit integer where 1 indicates a filter pass and 0 indicates a filter fail. This is the [RAIL_AddrConfig_t::matchTable](#) field and is used during filtering. For common simple configurations, two defines are provided with the truth tables as shown below. The first is [ADDRCONFIG_MATCH_TABLE_SINGLE_FIELD](#), which can be used if only using one address field (either field). If using two fields and want to force in the same address entry in each field, use the second define: [ADDRCONFIG_MATCH_TABLE_DOUBLE_FIELD](#). For more complex systems, create a valid custom table.

Note

- Address filtering does not function reliably with PHYs that use a data rate greater than 500 kbps. If this is a requirement, filtering must currently be done by the application.

Modules

[RAIL_AddrConfig_t](#)

Typedefs

typedef uint8_t	RAIL_AddrFilterMask_t
A bitmask representation of which 4 filters passed for each ADDRCONFIG_MAX_ADDRESS_FIELDS when filtering has completed successfully.	

Functions

RAIL_Status_t	RAIL_ConfigAddressFilter (RAIL_Handle_t railHandle, const RAIL_AddrConfig_t *addrConfig) Configure address filtering.
bool	RAIL_EnableAddressFilter (RAIL_Handle_t railHandle, bool enable) Enable address filtering.
bool	RAIL_IsAddressFilterEnabled (RAIL_Handle_t railHandle) Return whether address filtering is currently enabled.
RAIL_Status_t	RAIL_ResetAddressFilter (RAIL_Handle_t railHandle) Reset the address filtering configuration.
RAIL_Status_t	RAIL_SetAddressFilterAddress (RAIL_Handle_t railHandle, uint8_t field, uint8_t index, const uint8_t *value, bool enable) Set an address for filtering in hardware.
RAIL_Status_t	RAIL_SetAddressFilterAddressMask (RAIL_Handle_t railHandle, uint8_t field, const uint8_t *bitMask) Set an address bit mask for filtering in hardware.
RAIL_Status_t	RAIL_EnableAddressFilterAddress (RAIL_Handle_t railHandle, bool enable, uint8_t field, uint8_t index) Enable address filtering for the specified address.

Macros

#define	ADDRCONFIG_MATCH_TABLE_SINGLE_FIELD (0x1FFFFFFE) A default address filtering match table for configurations that use only one address field.
#define	ADDRCONFIG_MATCH_TABLE_DOUBLE_FIELD (0x1041040) A default address filtering match table for configurations that use two address fields and want to match the same index in each.
#define	ADDRCONFIG_MAX_ADDRESS_FIELDS (2) The maximum number of address fields that can be used by the address filtering logic.

Typedef Documentation

RAIL_AddrFilterMask_t

```
typedef uint8_t RAIL_AddrFilterMask_t
```

A bitmask representation of which 4 filters passed for each [ADDRCONFIG_MAX_ADDRESS_FIELDS](#) when filtering has completed successfully.

It's layout is: | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | |-----+-----+-----+-----+-----+-----+-----+-----|
—+-----| | Second Address Field Nibble | First Address Field Nibble | | Addr 3 | Addr 2 | Addr 1 | Addr 0 | Addr 3 | Addr 2 |
Addr 1 | Addr 0 | | match | match | match | match | match | match | match | match | |-----+-----+-----+-----+-----+-----|
+-----+-----+-----|

Note

-

This information is valid in [RAIL_IEEE802154_Address_t](#) on all platforms, but is only valid in [RAIL_RxPacketInfo_t](#) on platforms where [RAIL_SUPPORTS_ADDR_FILTER_MASK](#) is true.

Definition at line 3790 of file `common/rail_types.h`

Function Documentation

RAIL_ConfigAddressFilter

```
RAIL_Status_t RAIL_ConfigAddressFilter (RAIL_Handle_t railHandle, const RAIL_AddrConfig_t * addrConfig)
```

Configure address filtering.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	addrConfig	A pointer to the configuration structure, which defines how addresses are set up in your packets. May be NULL to reset address filtering.

Returns

- Status code indicating success of the function call.

You must call this function to set up address filtering. You may call it multiple times but all previous information is wiped out each time you call and any configured addresses must be reset.

Definition at line 4688 of file `common/rail.h`

RAIL_EnableAddressFilter

```
bool RAIL_EnableAddressFilter (RAIL_Handle_t railHandle, bool enable)
```

Enable address filtering.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	An argument to indicate whether or not to enable address filtering.

Returns

- true if address filtering was enabled to start with and false otherwise.

Only allow packets through that pass the current address filtering configuration. This does not reset or change the configuration so you can set that up before turning on this feature.

Definition at line 4704 of file `common/rail.h`

RAIL_IsAddressFilterEnabled

```
bool RAIL_IsAddressFilterEnabled (RAIL_Handle_t railHandle)
```

Return whether address filtering is currently enabled.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if address filtering is enabled and false otherwise.

Definition at line 4712 of file common/rail.h

RAIL_ResetAddressFilter

```
RAIL_Status_t RAIL_ResetAddressFilter (RAIL_Handle_t railHandle)
```

Reset the address filtering configuration.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

Resets all structures related to address filtering. This does not disable address filtering. It leaves the radio in a state where no packets pass filtering.

Definition at line 4724 of file common/rail.h

RAIL_SetAddressFilterAddress

```
RAIL_Status_t RAIL_SetAddressFilterAddress (RAIL_Handle_t railHandle, uint8_t field, uint8_t index, const uint8_t * value, bool enable)
```

Set an address for filtering in hardware.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	field	Indicates an address field for this address.
[in]	index	Indicates a match entry for this address for a given field.
[in]	value	A pointer to the address data. This must be at least as long as the size specified in RAIL_ConfigAddressFilter() . The first byte, value[0], will be compared to the first byte received over the air for this address field.
[in]	enable	A boolean to indicate whether this address should be enabled immediately.

Returns

- Status code indicating success of the function call.

This function loads the given address into hardware for filtering and starts filtering if you set the enable parameter to true. Otherwise, call [RAIL_EnableAddressFilterAddress\(\)](#) to turn it on later.

Definition at line 4745 of file common/rail.h

RAIL_SetAddressFilterAddressMask

```
RAIL_Status_t RAIL_SetAddressFilterAddressMask (RAIL_Handle_t railHandle, uint8_t field, const uint8_t * bitMask)
```

Set an address bit mask for filtering in hardware.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	field	Indicates an address field for this address bit mask.
[in]	bitMask	A pointer to the address bitmask. This must be at least as long as the size specified in RAIL_ConfigAddressFilter() . The first byte, bitMask[0], will be applied to the first byte received over the air for this address field. Bits set to 1 in the bit mask indicate which bit positions in the incoming packet to compare against the stored addresses during address filtering. Bits set to 0 indicate which bit positions to ignore in the incoming packet during address filtering. This bit mask is applied to all address entries.

Returns

- Status code indicating success of the function call.

This function loads the given address bit mask into hardware for use when address filtering is enabled. All bits in the stored address bit mask are set to 1 during hardware initialization and when either [RAIL_ConfigAddressFilter\(\)](#) or [RAIL_ResetAddressFilter\(\)](#) are called.

Note

- This feature/API is not supported on the EFR32xG21. Use the compile time symbol [RAIL_SUPPORTS_ADDR_FILTER_ADDRESS_BIT_MASK](#) or the runtime call [RAIL_SupportsAddrFilterAddressBitMask\(\)](#) to check whether the platform supports this feature.

Definition at line 4777 of file common/rail.h

RAIL_EnableAddressFilterAddress

```
RAIL_Status_t RAIL_EnableAddressFilterAddress (RAIL_Handle_t railHandle, bool enable, uint8_t field, uint8_t index)
```

Enable address filtering for the specified address.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	An argument to indicate whether or not to enable address filtering.
[in]	field	Indicates an address for the address.
[in]	index	Indicates a match entry in the given field you want to enable.

Returns

- Status code indicating success of the function call.

Definition at line 4791 of file common/rail.h

RAIL_AddrConfig_t

A structure to configure the address filtering functionality in RAIL.

Public Attributes

uint8_t	offsets	A list of the start offsets for each field.
uint8_t	sizes	A list of the address field sizes.
uint32_t	matchTable	The truth table to determine how the two fields combine to create a match.

Public Attribute Documentation

offsets

```
uint8_t RAIL_AddrConfig_t::offsets[(2)]
```

A list of the start offsets for each field.

These offsets are specified relative to the previous field's end. For the first field, it is relative to either the beginning of the packet or the end of the frame type byte if frame type decoding is enabled. If a field is unused, it's offset should be set to 0.

Definition at line 3746 of file common/rail_types.h

sizes

```
uint8_t RAIL_AddrConfig_t::sizes[(2)]
```

A list of the address field sizes.

These sizes are specified in bytes from 0 to 8. If you choose a size of 0, this field is effectively disabled.

Definition at line 3754 of file common/rail_types.h

matchTable

```
uint32_t RAIL_AddrConfig_t::matchTable
```

The truth table to determine how the two fields combine to create a match.

For detailed information about how this truth table is formed, see the detailed description of [Address Filtering](#).

For simple predefined configurations use the following defines.

- ADDRCONFIG_MATCH_TABLE_SINGLE_FIELD
 - For filtering that only uses a single address field.
- ADDRCONFIG_MATCH_TABLE_DOUBLE_FIELD for two field filtering where you

- For filtering that uses two address fields in a configurations where you want the following logic `((Field_0, Index_0) && (Field_1, Index_0)) || ((Field_0, Index_1) && (Field_1, Index_1)) || ...`

Definition at line 3770 of file `common/rail_types.h`

Packet Information

Packet Information

APIs to get information about received packets.

After receiving a packet, RAIL will trigger a [RAIL_EVENT_RX_PACKET_RECEIVED](#) event. At that point, there is a variety of information available to the application about the received packet. The following example code assumes that the [RAIL_RX_OPTION_REMOVE_APPENDED_INFO](#) is not used, and the application wants as much data about the packet as possible.

```
// Get all information about a received packet.
RAIL_Status_t status;
RAIL_RxPacketInfo_t rxInfo;
RAIL_RxPacketDetails_t rxDetails;
RAIL_RxPacketHandle_t rxHandle
    = RAIL_GetRxPacketInfo(railHandle, RAIL_RX_PACKET_HANDLE_NEWEST, &rxInfo);
assert(rxHandle != RAIL_RX_PACKET_HANDLE_INVALID);
status = RAIL_GetRxPacketDetailsAlt(railHandle, rxHandle, &rxDetails);
assert(status == RAIL_STATUS_NO_ERROR);
if (rxDetails.timeReceived.timePosition == RAIL_PACKET_TIME_INVALID) {
    return; // No timestamp available for this packet
}
// CRC_BYTES only needs to be added when not using RAIL_RX_OPTION_STORE_CRC
rxDetails.timeReceived.totalPacketBytes = rxInfo.packetBytes + CRC_BYTES;
// Choose the function which gives the desired timestamp
status = RAIL_GetRxTimeFrameEndAlt(railHandle, &rxDetails);
assert(status == RAIL_STATUS_NO_ERROR);
// Now all fields of rxInfo and rxDetails have been populated correctly
```

Functions

RAIL_RxPacketHandle_t	RAIL_GetRxPacketInfo (RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, RAIL_RxPacketInfo_t *pPacketInfo) Get basic information about a pending or received packet.
RAIL_Status_t	RAIL_GetRxIncomingPacketInfo (RAIL_Handle_t railHandle, RAIL_RxPacketInfo_t *pPacketInfo) Get information about the live incoming packet (if any).
void	RAIL_CopyRxPacket (uint8_t *pDest, const RAIL_RxPacketInfo_t *pPacketInfo) Copy a full packet to a user-specified contiguous buffer.
RAIL_Status_t	RAIL_GetRxPacketDetails (RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, RAIL_RxPacketDetails_t *pPacketDetails) Get detailed information about a received packet.
RAIL_Status_t	RAIL_GetRxPacketDetailsAlt (RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, RAIL_RxPacketDetails_t *pPacketDetails) Get detailed information about a received packet.
RAIL_Status_t	RAIL_GetRxTimePreambleStart (RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime) Adjust a RAIL RX timestamp to refer to the start of the preamble.
RAIL_Status_t	RAIL_GetRxTimePreambleStartAlt (RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails) Adjust a RAIL RX timestamp to refer to the start of the preamble.

- RAIL_Status_t

RAIL_GetRxTimeSyncWordEnd

(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)

Adjust a RAIL RX timestamp to refer to the end of the sync word.
- RAIL_Status_t

RAIL_GetRxTimeSyncWordEndAlt

(RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails)

Adjust a RAIL RX timestamp to refer to the end of the sync word.
- RAIL_Status_t

RAIL_GetRxTimeFrameEnd

(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)

Adjust a RAIL RX timestamp to refer to the end of frame.
- RAIL_Status_t

RAIL_GetRxTimeFrameEndAlt

(RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails)

Adjust a RAIL RX timestamp to refer to the end of frame.

Function Documentation

RAIL_GetRxPacketInfo

RAIL_RxPacketHandle_t RAIL_GetRxPacketInfo (RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, RAIL_RxPacketInfo_t * pPacketInfo)

Get basic information about a pending or received packet.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	packetHandle	A packet handle for the unreleased packet as returned from a previous call, or sentinel values RAIL_RX_PACKET_HANDLE_OLDEST , RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE or RAIL_RX_PACKET_HANDLE_NEWEST .
[out]	pPacketInfo	A non-NULL pointer to a RAIL_RxPacketInfo_t to store info for the requested packet.

Returns

- The packet handle for the requested packet: if packetHandle was one of the sentinel values, returns the actual packet handle for that packet, otherwise returns packetHandle. It may return [RAIL_RX_PACKET_HANDLE_INVALID](#) to indicate an error.

This function can be used in any RX mode. It does not free any receive FIFO or internal receive resources. If used in receive [RAIL_DataMethod_t::FIFO_MODE](#), the value in [RAIL_RxPacketInfo_t::packetBytes](#) will only return the data remaining in the FIFO. Any data read via earlier calls to [RAIL_ReadRxFifo\(\)](#) is not included.

Note

- When getting information about an arriving packet that is not yet complete, (i.e., pPacketInfo->packetStatus == [RAIL_RX_PACKET_RECEIVING](#)), keep in mind its data is highly suspect because it has not yet passed any CRC integrity checking. Also note that the packet could be aborted, canceled, or fail momentarily, invalidating its data in Packet mode. Furthermore, there is a small chance towards the end of packet reception that the filled-in [RAIL_RxPacketInfo_t](#) could include not only packet data received so far, but also some raw radio-appended info detail bytes that RAIL's packet-completion processing will subsequently deal with. It's up to the application to know its packet format well enough to avoid confusing such info as packet data.

Definition at line 3965 of file `common/rail.h`

RAIL_GetRxIncomingPacketInfo

RAIL_Status_t RAIL_GetRxIncomingPacketInfo (RAIL_Handle_t railHandle, RAIL_RxPacketInfo_t * pPacketInfo)

Get information about the live incoming packet (if any).

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	pPacketInfo	A non-NULL pointer to store RAIL_RxPacketInfo_t for the incoming packet.

Differs from [RAIL_GetRxPacketInfo\(\)](#) by only returning information about a packet actively being received, something which even the [RAIL_RX_PACKET_HANDLE_NEWEST](#) may not represent if there are completed but unprocessed packets in the receive FIFO.

Returns

- Status code indicating success of the function call.

This function can only be called from callback context, e.g., when handling [RAIL_EVENT_RX_FILTER_PASSED](#) or [RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#). It must not be used with receive [RAIL_DataMethod_t::FIFO_MODE](#) if any portion of an incoming packet has already been extracted from the receive FIFO.

Note

- The incomplete data of an arriving packet is highly suspect because it has not yet passed any CRC integrity checking. Also note that the packet could be aborted, canceled, or fail momentarily, invalidating its data in Packet mode. Furthermore, there is a small chance towards the end of packet reception that the filled-in [RAIL_RxPacketInfo_t](#) could include not only packet data received so far, but also some raw radio-appended info detail bytes that RAIL's packet-completion processing will subsequently deal with. It's up to the application to know its packet format well enough to avoid confusing such info as packet data.

Definition at line 3999 of file `common/rail.h`

RAIL_CopyRxPacket

```
static void RAIL_CopyRxPacket (uint8_t * pDest, const RAIL\_RxPacketInfo\_t * pPacketInfo)
```

Copy a full packet to a user-specified contiguous buffer.

Parameters

[out]	pDest	A non-NULL application-provided pointer to a buffer of at least <code>pPacketInfo->packetBytes</code> in size to store the packet data contiguously. This buffer must never overlay RAIL's receive FIFO buffer. Exactly <code>pPacketInfo->packetBytes</code> of packet data will be written into it.
[in]	pPacketInfo	A non-NULL pointer to the RAIL_RxPacketInfo_t for the requested packet.

Note

- This is a convenience helper function, which is intended to be expedient. As a result, it does not check the validity of its arguments, so don't pass either as NULL, and don't pass a pDest pointer to a buffer that's too small for the packet's data.
- If only a portion of the packet is needed, use [RAIL_PeekRxPacket\(\)](#) instead.

Definition at line 4023 of file `common/rail.h`

RAIL_GetRxPacketDetails

```
RAIL_Status_t RAIL_GetRxPacketDetails (RAIL\_Handle\_t railHandle, RAIL\_RxPacketHandle\_t packetHandle, RAIL\_RxPacketDetails\_t * pPacketDetails)
```

Get detailed information about a received packet.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	packetHandle	A packet handle for the unreleased packet as returned from a previous call to RAIL_GetRxPacketInfo() or RAIL_HoldRxPacket() , or sentinel values RAIL_RX_PACKET_HANDLE_OLDEST , RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE or RAIL_RX_PACKET_HANDLE_NEWEST .
[inout]	pPacketDetails	An application-provided non-NULL pointer to store RAIL_RxPacketDetails_t for the requested packet. For RAIL_RxPacketStatus_t RAIL_RX_PACKET_READY_ packets, the timeReceived fields totalPacketBytes and timePosition must be initialized prior to each call: - totalPacketBytes with the total number of bytes of the received packet for RAIL to use when calculating the specified timestamp. This should account for all bytes received over the air after the Preamble and Sync word(s), including CRC bytes. - timePosition with a RAIL_PacketTimePosition_t value specifying the packet position to put in the timeReceived field on return. This field will also be updated with the actual position corresponding to the timeReceived value filled in.

This function can be used in any RX mode; it does not free any receive FIFO or internal receive resources.

Returns

- [RAIL_STATUS_NO_ERROR](#) if pPacketDetails was filled in, or an appropriate error code otherwise.

Note

- Certain details are always available, while others are only available if the [RAIL_RxOptions_t](#)[RAIL_RX_OPTION_REMOVE_APPENDED_INFO](#) option is not in effect and the received packet's [RAIL_RxPacketStatus_t](#) is among the [RAIL_RX_PACKET_READY_](#) set. See [RAIL_RxPacketDetails_t](#) for clarification.
- Consider using [RAIL_GetRxPacketDetailsAlt](#) for smaller code size.

Definition at line 4071 of file common/rail.h

RAIL_GetRxPacketDetailsAlt

```
RAIL_Status_t RAIL_GetRxPacketDetailsAlt (RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, RAIL_RxPacketDetails_t * pPacketDetails)
```

Get detailed information about a received packet.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	packetHandle	A packet handle for the unreleased packet as returned from a previous call to RAIL_GetRxPacketInfo() or RAIL_HoldRxPacket() , or sentinel values RAIL_RX_PACKET_HANDLE_OLDEST , RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE or RAIL_RX_PACKET_HANDLE_NEWEST .
[out]	pPacketDetails	A non-NULL application-provided pointer to store RAIL_RxPacketDetails_t for the requested packet. For RAIL_RxPacketStatus_t RAIL_RX_PACKET_READY_ packets, the timeReceived field packetTime will be populated with a timestamp corresponding to a default location in the packet. The timeReceived field timePosition will be populated with a RAIL_PacketTimePosition_t value specifying that default packet location. Call RAIL_GetRxTimePreambleStart() , RAIL_GetRxTimeSyncWordEnd() , or RAIL_GetRxTimeFrameEnd() to adjust that timestamp for different locations in the packet.

This function can be used in any RX mode; it does not free any receive FIFO or receive internal resources.

Returns

- [RAIL_STATUS_NO_ERROR](#) if pPacketDetails was filled in, or an appropriate error code otherwise.

This alternative API allows for smaller code size by deadstripping the timestamp adjustment algorithms which are not in use.

Note

- Certain details are always available, while others are only available if the [RAIL_RxOptions_tRAIL_RX_OPTION_REMOVE_APPENDED_INFO](#) option is not in effect and the received packet's [RAIL_RxPacketStatus_t](#) is among the RAIL_RX_PACKET_READY_ set. See [RAIL_RxPacketDetails_t](#) for clarification.
- This function should be called soon (no more than a minute) after packet reception for the packet timestamp information to be valid.

Definition at line 4111 of file common/rail.h

RAIL_GetRxTimePreambleStart

```
RAIL_Status_t RAIL_GetRxTimePreambleStart (RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t * pPacketTime)
```

Adjust a RAIL RX timestamp to refer to the start of the preamble.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	totalPacketBytes	The total number of bytes of the received packet for RAIL to use when calculating the specified timestamp. This should account for all bytes received over the air after the Preamble and Sync word(s), including CRC bytes.
[inout]	pPacketTime	A pointer to the time that was returned in the RAIL_PacketTimeStamp_t::packetTime field of RAIL_RxPacketDetails_t::timeReceived from a previous call to RAIL_GetRxPacketDetailsAlt for this same packet. After this function, the time at that location will be updated with the time that the preamble for this packet started on air. Must be non-NULL.

Returns

- [RAIL_STATUS_NO_ERROR](#) if pPacketTime was successfully calculated, or an appropriate error code otherwise.

Call this API while the given railHandle is active, or it will return an error code of [RAIL_STATUS_INVALID_STATE](#). Note that this API may return incorrect timestamps when sub-phys are in use. Prefer [RAIL_GetRxTimePreambleStartAlt](#) in those situations. See [RAIL_RxPacketDetails_t::subPhyId](#) for more details.

Definition at line 4138 of file common/rail.h

RAIL_GetRxTimePreambleStartAlt

```
RAIL_Status_t RAIL_GetRxTimePreambleStartAlt (RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t * pPacketDetails)
```

Adjust a RAIL RX timestamp to refer to the start of the preamble.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	A non-NULL pointer to the details that were returned from a previous call to RAIL_GetRxPacketDetailsAlt() for this same packet. The application must update the timeReceived field totalPacketBytes to be the total number of bytes of the received packet for RAIL to use when calculating the specified timestamp. This should account for all bytes received over the air after the Preamble and Sync word(s), including CRC bytes. After this function, the timeReceived field packetTime will be updated with the time that the preamble for this packet started on air.

Returns

- [RAIL_STATUS_NO_ERROR](#) if the packet time was successfully calculated, or an appropriate error code otherwise.

Definition at line 4157 of file common/rail.h

RAIL_GetRxTimeSyncWordEnd

```
RAIL_Status_t RAIL_GetRxTimeSyncWordEnd (RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t * pPacketTime)
```

Adjust a RAIL RX timestamp to refer to the end of the sync word.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	totalPacketBytes	The total number of bytes of the received packet for RAIL to use when calculating the specified timestamp. This should account for all bytes received over the air after the Preamble and Sync word(s), including CRC bytes.
[inout]	pPacketTime	A pointer to the time that was returned in the RAIL_PacketTimeStamp_t::packetTime field of RAIL_RxPacketDetails_t::timeReceived from a previous call to RAIL_GetRxPacketDetailsAlt for this same packet. After this function, the time at that location will be updated with the time that the sync word for this packet finished on air. Must be non-NULL.

Returns

- [RAIL_STATUS_NO_ERROR](#) if pPacketTime was successfully calculated, or an appropriate error code otherwise.

Call this API while the given railHandle is active, or it will return an error code of [RAIL_STATUS_INVALID_STATE](#). Note that this API may return incorrect timestamps when sub-phys are in use. Prefer [RAIL_GetRxTimePreambleStartAlt](#) in those situations. See [RAIL_RxPacketDetails_t::subPhyId](#) for more details.

Definition at line 4183 of file common/rail.h

RAIL_GetRxTimeSyncWordEndAlt

```
RAIL_Status_t RAIL_GetRxTimeSyncWordEndAlt (RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t * pPacketDetails)
```

Adjust a RAIL RX timestamp to refer to the end of the sync word.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	A non-NULL pointer to the details that were returned from a previous call to RAIL_GetRxPacketDetailsAlt() for this same packet. The application must update the timeReceived field totalPacketBytes to be the total number of bytes of the received packet for RAIL to use when calculating the specified timestamp. This should account for all bytes received over the air after the Preamble and Sync word(s), including CRC bytes. After this function, the timeReceived field packetTime will be updated with the time that the sync word for this packet finished on air.

Returns

- [RAIL_STATUS_NO_ERROR](#) if the packet time was successfully calculated, or an appropriate error code otherwise.

Definition at line 4202 of file common/rail.h

RAIL_GetRxTimeFrameEnd

```
RAIL_Status_t RAIL_GetRxTimeFrameEnd (RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t * pPacketTime)
```

Adjust a RAIL RX timestamp to refer to the end of frame.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	totalPacketBytes	The total number of bytes of the received packet for RAIL to use when calculating the specified timestamp. This should account for all bytes received over the air after the Preamble and Sync word(s), including CRC bytes.
[inout]	pPacketTime	A pointer to the time that was returned in the RAIL_PacketTimeStamp_t::packetTime field of RAIL_RxPacketDetails_t::timeReceived from a previous call to RAIL_GetRxPacketDetailsAlt for this same packet. After this function, the time at that location will be updated with the time that this packet finished on air. Must be non-NULL.

Returns

- [RAIL_STATUS_NO_ERROR](#) if pPacketTime was successfully calculated, or an appropriate error code otherwise.

Call this API while the given railHandle is active, or it will return an error code of [RAIL_STATUS_INVALID_STATE](#). Note that this API may return incorrect timestamps when sub-phys are in use. Prefer [RAIL_GetRxTimePreambleStartAlt](#) in those situations. See [RAIL_RxPacketDetails_t::subPhyId](#) for more details.

Definition at line 4228 of file `common/rail.h`

RAIL_GetRxTimeFrameEndAlt

```
RAIL_Status_t RAIL_GetRxTimeFrameEndAlt (RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t * pPacketDetails)
```

Adjust a RAIL RX timestamp to refer to the end of frame.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	A non-NULL pointer to the details that were returned from a previous call to RAIL_GetRxPacketDetailsAlt() for this same packet. The application must update the timeReceived field totalPacketBytes to be the total number of bytes of the received packet for RAIL to use when calculating the specified timestamp. This should account for all bytes received over the air after the Preamble and Sync word(s), including CRC bytes. After this function, the timeReceived field packetTime will be updated with the time that the packet finished on air.

Returns

- [RAIL_STATUS_NO_ERROR](#) if the packet time was successfully calculated, or an appropriate error code otherwise.

Definition at line 4247 of file `common/rail.h`

Retiming

Retiming

EFR32-specific retiming capability.

The EFR product families have many digital and analog modules that can run in parallel with a radio. These combinations can cause interference and degradation on the radio RX sensitivity. Retiming can modify the clocking of the digital modules to reduce the interference.

Enumerations

```
enum RAIL_RetimeOptions_t {  
    RAIL_RETIME_OPTION_HFXO_SHIFT = 0  
    RAIL_RETIME_OPTION_HFRCO_SHIFT = 1  
    RAIL_RETIME_OPTION_DCDC_SHIFT = 2  
    RAIL_RETIME_OPTION_LCD_SHIFT = 3  
}  
Retiming options bit shifts.
```

Functions

RAIL_Status_t	RAIL_ConfigRetimeOptions(RAIL_Handle_t railHandle, RAIL_RetimeOptions_t mask, RAIL_RetimeOptions_t options) Configure retiming options.
RAIL_Status_t	RAIL_GetRetimeOptions(RAIL_Handle_t railHandle, RAIL_RetimeOptions_t *pOptions) Get the currently configured retiming option.
RAIL_Status_t	RAIL_ChangedDcdc(void) Indicate that the DCDC peripheral bus clock enable has changed allowing RAIL to react accordingly.

Macros

#define	RAIL_RETIME_OPTION_HFXO (1U << RAIL_RETIME_OPTION_HFXO_SHIFT) An option to configure HFXO retiming.
#define	RAIL_RETIME_OPTION_HFRCO (1U << RAIL_RETIME_OPTION_HFRCO_SHIFT) An option to configure HFRCO retiming.
#define	RAIL_RETIME_OPTION_DCDC (1U << RAIL_RETIME_OPTION_DCDC_SHIFT) An option to configure DCDC retiming.
#define	RAIL_RETIME_OPTION_LCD (1U << RAIL_RETIME_OPTION_LCD_SHIFT) An option to configure LCD retiming.
#define	RAIL_RETIME_OPTIONS_NONE 0x0U A value representing no retiming options.
#define	RAIL_RETIME_OPTIONS_ALL 0xFFU A value representing all retiming options.

Enumeration Documentation

RAIL_RetimeOptions_t

RAIL_RetimeOptions_t

Retiming options bit shifts.

Enumerator	
RAIL_RETIME_OPTION_HFXO_SHIFT	Shift position of RAIL_RETIME_OPTION_HFXO bit.
RAIL_RETIME_OPTION_HFRCO_SHIFT	Shift position of RAIL_RETIME_OPTION_HFRCO bit.
RAIL_RETIME_OPTION_DCDC_SHIFT	Shift position of RAIL_RETIME_OPTION_DCDC bit.
RAIL_RETIME_OPTION_LCD_SHIFT	Shift position of RAIL_RETIME_OPTION_LCD bit.

Definition at line 5837 of file `common/rail_types.h`

Function Documentation

RAIL_ConfigRetimeOptions

RAIL_Status_t RAIL_ConfigRetimeOptions ([RAIL_Handle_t](#) railHandle, [RAIL_RetimeOptions_t](#) mask, [RAIL_RetimeOptions_t](#) options)

Configure retiming options.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	mask	A bitmask containing which options should be modified.
[in]	options	A bitmask containing desired configuration settings. Bit positions for each option are found in the RAIL_RetimeOptions_t .

Returns

- Status code indicating success of the function call.

Definition at line 6572 of file `common/rail.h`

RAIL_GetRetimeOptions

RAIL_Status_t RAIL_GetRetimeOptions ([RAIL_Handle_t](#) railHandle, [RAIL_RetimeOptions_t](#) * pOptions)

Get the currently configured retiming option.

Parameters

[in]	railHandle	A handle of RAIL instance.
[out]	pOptions	A pointer to configured retiming options bitmask indicating which are enabled.

Returns

- Status code indicating success of the function call.

Definition at line 6584 of file common/rail.h

RAIL_ChangedDcdc

```
RAIL_Status_t RAIL_ChangedDcdc (void )
```

Indicate that the DCDC peripheral bus clock enable has changed allowing RAIL to react accordingly.

Parameters

N/A		

Returns

- Status code indicating success of the function call.

Note

- This should be called after DCDC has been enabled or disabled.

Definition at line 6595 of file common/rail.h

Sleep

Sleep

These APIs help when putting the system to an EM2/EM3/EM4 sleep states where the high frequency clock is disabled.

The RAIL library has its own timebase and the ability to schedule operations into the future. When going to any power mode that disables the HF clock used for the radio (EM2/EM3/EM4), it is important that this timebase is synchronized to a running LFCLK and the chip is set to wake up before the next scheduled event. If RAIL has not been configured to use the power manager, [RAIL_Sleep\(\)](#) and [RAIL_Wake\(\)](#) must be called for performing this synchronization. If RAIL has been configured to use the power manager, [RAIL_InitPowerManager\(\)](#), it will automatically perform timer synchronization based on the selected [RAIL_TimerSyncConfig_t](#). Calls to [RAIL_Sleep\(\)](#) and [RAIL_Wake\(\)](#) are unsupported in such a scenario.

Following example code snippets demonstrate synchronizing the timebase with and without timer synchronization:

Sleep with timer synchronization:

When sleeping with timer synchronization, you must first get the required LFCLK up and running and leave it running across sleep so that the high frequency clock that drives the RAIL time base can be synchronized to it. The [RAIL_Sleep\(\)](#) API will also set up a wake event on the timer to wake up `wakeupProcessTime` before the next timer event so that it can run successfully. See the [EFR32](#) sections on Low-Frequency Clocks and RAIL Timer Synchronization for more setup details.

This is useful when maintaining packet timestamps across sleep or use the scheduled RX/TX APIs while sleeping in between. It does take more time and code to do the synchronization. If your application does not need this, it should be avoided.

Example (without Power Manager):

```
#include "rail.h"

extern RAIL_Handle_t railHandle;
// Wakeup time for your crystal/board/chip combination
extern uint32_t wakeupProcessTime;

void main(void)
{
    RAIL_Status_t status;
    bool shouldSleep = false;

    // This function depends on your board/chip but it must enable the LFCLK
    // you intend to use for RTCC sync before we configure sleep as that function
    // will attempt to auto detect the clock.
    BoardSetupLFCLK();

    // Configure sleep for timer synchronization
    RAIL_TimerSyncConfig_t timerSyncConfig = RAIL_TIMER_SYNC_DEFAULT;
    status = RAIL_ConfigSleepAlt(railHandle, &timerSyncConfig);
    assert(status == RAIL_STATUS_NO_ERROR);

    // Application main loop
    while(1) {
        // ... do normal app stuff and set shouldSleep to true when we want to
        // sleep
        if (shouldSleep) {
            bool sleepAllowed = false;

            // Go critical to assess sleep decisions
            CORE_ENTER_CRITICAL();
            if (RAIL_Sleep(wakeupProcessTime, &sleepAllowed) != RAIL_STATUS_NO_ERROR) {
                printf("Error trying to go to sleep!");
                CORE_EXIT_CRITICAL();
                continue;
            }
            if (sleepAllowed) {
                // Go to sleep
            }
            // Wakeup and sync the RAIL timebase back up
            RAIL_Wake(0);
            CORE_EXIT_CRITICAL();
        }
    }
}
```

Example (with Power Manager):

```

#include "rail.h"
#include "sl_power_manager.h"

extern RAIL_Handle_t railHandle;

void main(void)
{
    RAIL_Status_t status;
    bool shouldSleep = false;

    // This function depends on your board/chip but it must enable the LFCLK
    // you intend to use for RTCC sync before we configure sleep as that function
    // will attempt to auto detect the clock.
    BoardSetupLFCLK();
    // Configure sleep for timer synchronization
    RAIL_TimerSyncConfig_t timerSyncConfig = RAIL_TIMER_SYNC_DEFAULT;
    status = RAIL_ConfigSleepAlt(railHandle, &timerSyncConfig);
    assert(status == RAIL_STATUS_NO_ERROR);
    // Initialize application-level power manager service
    sl_power_manager_init();
    // Initialize RAIL library's use of the power manager
    RAIL_InitPowerManager();

    // Application main loop
    while(1) {
        // ... do normal app stuff and set shouldSleep to true when we want to
        // sleep
        if (shouldSleep) {
            // Let the CPU go to sleep if the system allows it.
            sl_power_manager_sleep();
        }
    }
}

```

RAIL APIs such as, [RAIL_StartScheduledTx\(\)](#), [RAIL_ScheduleRx\(\)](#), [RAIL_SetTimer\(\)](#), [RAIL_SetMultiTimer\(\)](#) can be used to schedule periodic wakeups to perform a scheduled operation. The call to `sl_power_manager_sleep()` in the main loop ensures that the device sleeps until the scheduled operation is due. Upon completion, each instantaneous or scheduled RX/TX operation will indicate radio busy to the power manager to allow the application to service the RAIL event and perform subsequent operations before going to sleep. Therefore, it is important that the application idle the radio by either calling [RAIL_Idle\(\)](#) or [RAIL_YieldRadio\(\)](#). If the radio transitions to RX after an RX or TX operation, always call [RAIL_Idle\(\)](#) in order transition to a lower sleep state. If the radio transitions to idle after an RX or TX operation, [RAIL_YieldRadio\(\)](#) should suffice in indicating to the power manager that the radio is no longer busy and the device can sleep.

The following example shows scheduling periodic TX on getting a TX completion event:

```

void RAILCb_Event(RAIL_Handle_t railHandle, RAIL_Events_t events)
{
    // Omitting other event handlers
    if (events & RAIL_EVENTS_TX_COMPLETION) {
        // Schedule the next TX.
        RAIL_ScheduleTxConfig_t config = {
            .when = (RAIL_Time_t)parameters->startTime,
            .mode = (RAIL_TimeMode_t)parameters->startTimeMode
        };
        (void)RAIL_StartScheduledTx(railHandle, channel, 0, &config, NULL);
    }
}

```

Note

- The above code assumes that RAIL automatic state transitions after TX are idle. Use [RAIL_SetTxTransitions\(\)](#) to ensure the right state transitions are used. Radio must be idle for the device to enter EM2 or lower energy mode.

- When using the power manager, usage of `RAIL_YieldRadio()` in single protocol RAIL is similar to its usage in multiprotocol RAIL. See [Yielding the Radio](#) for more details.
- Back to back scheduled operations do not require an explicit call to `RAIL_YieldRadio()` if the radio transitions to idle.

Sleep without timer synchronization:

When sleeping without timer synchronization, you are free to enable only the LFCLKs and wake sources required by the application. RAIL will not attempt to configure any wake events and may miss anything that occurs over sleep.

This is useful when your application does not care about packet timestamps or scheduling operations accurately over sleep.

Example (without Power Manager):

```
#include "rail.h"

extern RAIL_Handle_t railHandle;

void main(void)
{
    RAIL_Status_t status;
    bool shouldSleep = false;

    // Configure sleep for no timer synchronization
    RAIL_TimerSyncConfig_t timerSyncConfig = {
        .sleep = RAIL_SLEEP_CONFIG_TIMERSYNC_DISABLED,
    };
    status = RAIL_ConfigSleepAlt(railHandle, &timerSyncConfig);
    assert(status == RAIL_STATUS_NO_ERROR);

    // Application main loop
    while(1) {
        // ... do normal app stuff and set shouldSleep to true when we want to
        // sleep
        if (shouldSleep) {
            bool sleepAllowed = false;
            uint32_t sleepTime = 0;

            // Go critical to assess sleep decisions
            CORE_ENTER_CRITICAL();
            if (RAIL_Sleep(0, &sleepAllowed) != RAIL_STATUS_NO_ERROR) {
                printf("Error trying to go to sleep!");
                CORE_EXIT_CRITICAL();
                continue;
            }
            if (sleepAllowed) {
                // Go to sleep and optionally update sleepTime to the correct value
                // in microseconds
            }
            // Wakeup and sync the RAIL timebase back up
            RAIL_Wake(sleepTime);
            CORE_EXIT_CRITICAL();
        }
    }
}
```

Example (with Power Manager):

```
#include "rail.h"
#include "sl_power_manager.h"

extern RAIL_Handle_t railHandle;

void main(void)
{
    RAIL_Status_t status;
    bool shouldSleep = false;

    // This function depends on your board/chip but it must enable the LFCLK
    // you intend to use for RTCC sync before we configure sleep as that function
    // will attempt to auto detect the clock.
    BoardSetupLFCLK();
    // Configure sleep for no timer synchronization
    RAIL_TimerSyncConfig_t timerSyncConfig = {
        sleep = RAIL_SLEEP_CONFIG_TIMERSYNC_DISABLED,
    };
    status = RAIL_ConfigSleepAlt(railHandle, &timerSyncConfig);
    assert(status == RAIL_STATUS_NO_ERROR);
    // Initialize application-level power manager service
    sl_power_manager_init();
    // Initialize RAIL library's use of the power manager
    RAIL_InitPowerManager();

    // Application main loop
    while(1) {
        // ... do normal app stuff and set shouldSleep to true when we want to
        // sleep
        if (shouldSleep) {
            // Let the CPU go to sleep if the system allows it.
            sl_power_manager_sleep();
        }
    }
}
```

Modules

[RAIL_TimerSyncConfig_t](#)

[EFR32xG2x](#)

[S1xx3xx](#)

Enumerations

```
enum    RAIL\_SleepConfig\_t {
        RAIL_SLEEP_CONFIG_TIMERSYNC_DISABLED = 0
        RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED = 1
    }
    The configuration.
```

Functions

void [RAILCb_ConfigSleepTimerSync](#)(RAIL_TimerSyncConfig_t *timerSyncConfig)
Configure RAIL timer synchronization.

[RAIL_Status_t](#) [RAIL_ConfigSleep](#)(RAIL_Handle_t railHandle, RAIL_SleepConfig_t sleepConfig)
Initialize RAIL timer synchronization.

[RAIL_Status_t](#) [RAIL_ConfigSleepAlt](#)(RAIL_Handle_t railHandle, const RAIL_TimerSyncConfig_t *syncConfig)
Initialize RAIL timer synchronization.

RAIL_Status_t	RAIL_Sleep (uint16_t wakeupProcessTime, bool *deepSleepAllowed) Stop the RAIL timer(s) and prepare RAIL for sleep.
RAIL_Status_t	RAIL_Wake (RAIL_Time_t elapsedTime) Wake RAIL from sleep and restart the RAIL timer(s).
RAIL_Status_t	RAIL_InitPowerManager (void) Initialize RAIL Power Manager.
RAIL_Status_t	RAIL_DeinitPowerManager (void) Stop the RAIL Power Manager.

Macros

#define	RAIL_TIMER_SYNC_PRS_CHANNEL_DEFAULT (255U) Platform-agnostic value to use default PRS channel when configuring sleep.
#define	RAIL_TIMER_SYNC_RTCC_CHANNEL_DEFAULT (255U) Platform-agnostic vlaue to use default RTCC channel when configuring sleep.
#define	RAIL_TIMER_SYNC_DEFAULT undefined Default timer synchronization configuration.

Enumeration Documentation

RAIL_SleepConfig_t

RAIL_SleepConfig_t

The configuration.

Enumerator	
RAIL_SLEEP_CONFIG_TIMERSYNC_DISABLED	Disable timer sync before and after sleep.
RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED	Enable timer sync before and after sleep.

Definition at line 436 of file common/rail_types.h

Function Documentation

RAILCb_ConfigSleepTimerSync

void RAILCb_ConfigSleepTimerSync ([RAIL_TimerSyncConfig_t](#) * timerSyncConfig)

Configure RAIL timer synchronization.

Parameters

[inout]	timerSyncConfig	A pointer to the RAIL_TimerSyncConfig_t structure containing the configuration parameters for timer sync. The RAIL_TimerSyncConfig_t::sleep field is ignored in this call.

This function is optional to implement.

This function is called during [RAIL_ConfigSleep\(\)](#) to allow an application to configure the PRS and RTCC channels used for timer sync to values other than their defaults. The default channels are populated in timerSyncConfig and can be overwritten by the application. If this function is not implemented by the application, a default implementation from within the RAIL library will be used that simply maintains the default channel values in timerSyncConfig. For example:

```
void RAILCb_ConfigSleepTimerSync(RAIL_TimerSyncConfig_t *timerSyncConfig)
{
    timerSyncConfig->prsChannel = MY_TIMERSYNC_PRS_CHANNEL;
    timerSyncConfig->rtccChannel = MY_TIMERSYNC_RTCC_CHANNEL;
}
```

If an unsupported channel is selected by the application, [RAIL_ConfigSleep\(\)](#) will return [RAIL_STATUS_INVALID_PARAMETER](#).

Definition at line 1566 of file `common/rail.h`

RAIL_ConfigSleep

```
RAIL_Status_t RAIL_ConfigSleep (RAIL_Handle_t railHandle, RAIL_SleepConfig_t sleepConfig)
```

Initialize RAIL timer synchronization.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	sleepConfig	A sleep configuration.

Returns

- Status code indicating success of the function call.

Warnings

- As this function relies on PRS and SYSRTC access and RAIL is meant to run in TrustZone non-secure world, it is not supported if PRS or SYSRTC are configured as secure peripheral and sleepConfig is set to [RAIL_SleepConfig_t::RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED](#). It will return [RAIL_STATUS_INVALID_CALL](#).

Definition at line 1581 of file `common/rail.h`

RAIL_ConfigSleepAlt

```
RAIL_Status_t RAIL_ConfigSleepAlt (RAIL_Handle_t railHandle, const RAIL_TimerSyncConfig_t * syncConfig)
```

Initialize RAIL timer synchronization.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	syncConfig	A non-NULL pointer to the timer synchronization configuration.

Returns

- Status code indicating success of the function call.

The default structure used to enable timer synchronization across sleep is [RAIL_TIMER_SYNC_DEFAULT](#).

Warnings

- As this function relies on PRS and SYSRTC access and RAIL is meant to run in TrustZone non-secure world, it is not supported if PRS or SYSRTC are configured as secure peripheral and syncConfig->sleep is set to [RAIL_SleepConfig_t::RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED](#). It will return [RAIL_STATUS_INVALID_CALL](#).

Definition at line 1600 of file `common/rail.h`

RAIL_Sleep

RAIL_Status_t RAIL_Sleep (uint16_t wakeupProcessTime, bool * deepSleepAllowed)

Stop the RAIL timer(s) and prepare RAIL for sleep.

Parameters

[in]	wakeupProcessTime	Time in microseconds that the application and hardware need to recover from sleep state.
[out]	deepSleepAllowed	A pointer to boolean that will be set true if system can go to deep sleep or false if system must not go to deep sleep (EM2 or lower energy modes).

Returns

- Status code indicating success of the function call.

Warnings

- The active RAIL configuration must be idle to enable sleep.

Note

- This API must not be called if RAIL Power Manager is initialized.

Definition at line 1617 of file common/rail.h

RAIL_Wake

RAIL_Status_t RAIL_Wake ([RAIL_Time_t](#) elapsedTime)

Wake RAIL from sleep and restart the RAIL timer(s).

Parameters

[in]	elapsedTime	Add this sleep duration in microseconds to the RAIL timer(s) before restarting it(them).

Returns

- Status code indicating success of the function call.

If the timer sync was enabled by [RAIL_ConfigSleep\(\)](#), synchronize the RAIL timer(s) using an alternate timer. Otherwise, add elapsedTime to the RAIL timer(s).

Note

- This API must not be called if RAIL Power Manager is initialized.

Definition at line 1632 of file common/rail.h

RAIL_InitPowerManager

RAIL_Status_t RAIL_InitPowerManager (void)

Initialize RAIL Power Manager.

Parameters

N/A		

Returns

- Status code indicating success of the function call.

Note

- This function must be called only when the application is built and initialized with Power Manager plugin and when the radio is idle. RAIL will perform timer synchronization, upon transitioning from EM2 or lower to EM1 or higher energy mode or vice-versa, in the Power Manager EM transition callback.

Warnings

- Since EM transition callbacks are not called in a deterministic order, it is suggested to not call any RAIL time dependent APIs in an EM transition callback.
- As this function relies on EMU access and RAIL is meant to run in TrustZone non-secure world, it is not supported if EMU is configured as secure peripheral and it will return [RAIL_STATUS_INVALID_CALL](#).

Definition at line 1653 of file common/rail.h

RAIL_DeinitPowerManager

RAIL_Status_t RAIL_DeinitPowerManager (void)

Stop the RAIL Power Manager.

Parameters

N/A		

Returns

- Status code indicating success of the function call.

Note

- The active RAIL configuration must be idle to disable radio power manager and there should be no outstanding requirements by radio power manager.

Definition at line 1664 of file common/rail.h

RAIL_TimerSyncConfig_t

Channel values used to perform timer sync before and after sleep.

The default value of this structure is provided in the [RAIL_TIMER_SYNC_DEFAULT](#) macro.

Public Attributes

uint8_t	prsChannel	PRS Channel used for timer sync operations.
uint8_t	rtccChannel	RTCC Channel used for timer sync operations.
RAIL_SleepConfig_t	sleep	Whether to sync the timer before and after sleeping.

Public Attribute Documentation

prsChannel

uint8_t RAIL_TimerSyncConfig_t::prsChannel

PRS Channel used for timer sync operations.

Use [RAIL_TIMER_SYNC_PRS_CHANNEL_DEFAULT](#) or another suitable one.

Definition at line 467 of file `common/railTypes.h`

rtccChannel

uint8_t RAIL_TimerSyncConfig_t::rtccChannel

RTCC Channel used for timer sync operations.

Only applies to platforms where the RTCC used for timer sync has multiple channels. Use [RAIL_TIMER_SYNC_RTCC_CHANNEL_DEFAULT](#) or another suitable one.

Definition at line 473 of file `common/railTypes.h`

sleep

RAIL_SleepConfig_t RAIL_TimerSyncConfig_t::sleep

Whether to sync the timer before and after sleeping.

Definition at line 477 of file `common/railTypes.h`

EFR32xG2x

EFR32xG2x

EFR32xG2x-specific Sleeping.

Macros

```
#define RAILINT_TIMER_SYNC_PRS_CHANNEL_DEFAULT (7U)
Default PRS channel to use when configuring sleep.

#define RAILINT_TIMER_SYNC_RTCC_CHANNEL_DEFAULT (0U)
Default RTCC channel to use when configuring sleep.
```

Slxx3xx

Slxx3xx

Slxx3xx-specific Sleeping.

Macros

```
#define RAILINT_TIMER_SYNC_PRS_CHANNEL_DEFAULT (7U)
Default PRS channel to use when configuring sleep.

#define RAILINT_TIMER_SYNC_RTCC_CHANNEL_DEFAULT (0U)
Default RTCC channel to use when configuring sleep.
```

State Transitions

State Transitions

Modules

[RAIL_StateTiming_t](#)

[RAIL_StateTransitions_t](#)

[EFR32xG2x](#)

[Slxx3xx](#)

Enumerations

```
enum RAIL\_RadioState\_t {
    RAIL_RF_STATE_INACTIVE = 0u
    RAIL_RF_STATE_ACTIVE = (1u << 0)
    RAIL_RF_STATE_RX = (1u << 1)
    RAIL_RF_STATE_TX = (1u << 2)
    RAIL_RF_STATE_IDLE = (RAIL_RF_STATE_ACTIVE)
    RAIL_RF_STATE_RX_ACTIVE = (RAIL_RF_STATE_RX | RAIL_RF_STATE_ACTIVE)
    RAIL_RF_STATE_TX_ACTIVE = (RAIL_RF_STATE_TX | RAIL_RF_STATE_ACTIVE)
}
The state of the radio.
```

```
enum RAIL\_RadioStateDetail\_t {
    RAIL_RF_STATE_DETAIL_IDLE_STATE_SHIFT = 0
    RAIL_RF_STATE_DETAIL_RX_STATE_SHIFT = 1
    RAIL_RF_STATE_DETAIL_TX_STATE_SHIFT = 2
    RAIL_RF_STATE_DETAIL_TRANSITION_SHIFT = 3
    RAIL_RF_STATE_DETAIL_ACTIVE_SHIFT = 4
    RAIL_RF_STATE_DETAIL_NO_FRAMES_SHIFT = 5
    RAIL_RF_STATE_DETAIL_LBT_SHIFT = 6
}
The detailed state of the radio.
```

```
enum RAIL\_IdleMode\_t {
    RAIL_IDLE = 0u
    RAIL_IDLE_ABORT = 1u
    RAIL_IDLE_FORCE_SHUTDOWN = 2u
    RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS = 3u
}
An enumeration for the different types of supported idle modes.
```

Typedefs

```
typedef uint32_t RAIL\_TransitionTime\_t
Suitable type for the supported transition time range.
```

Functions

[RAIL_Status_t](#) [RAIL_SetRxTransitions](#)(RAIL_Handle_t railHandle, const RAIL_StateTransitions_t *transitions)

Configure RAIL
automatic state
transitions after RX.

RAIL_Status_t	RAIL_GetRxTransitions (RAIL_Handle_t railHandle, RAIL_StateTransitions_t *transitions) Get the current RAIL automatic state transitions after RX.
RAIL_Status_t	RAIL_SetTxTransitions (RAIL_Handle_t railHandle, const RAIL_StateTransitions_t *transitions) Configure RAIL automatic state transitions after TX.
RAIL_Status_t	RAIL_GetTxTransitions (RAIL_Handle_t railHandle, RAIL_StateTransitions_t *transitions) Get the current RAIL automatic state transitions after TX.
RAIL_Status_t	RAIL_SetNextTxRepeat (RAIL_Handle_t railHandle, const RAIL_TxRepeatConfig_t *repeatConfig) Set up automatic repeated transmits after the next transmit.
uint16_t	RAIL_GetTxPacketsRemaining (RAIL_Handle_t railHandle) Get the number of transmits remaining in a repeat operation.
RAIL_Status_t	RAIL_SetStateTiming (RAIL_Handle_t railHandle, RAIL_StateTiming_t *timings) Configure RAIL automatic state transition timing.
RAIL_Status_t	RAIL_Idle (RAIL_Handle_t railHandle, RAIL_IdleMode_t mode, bool wait) Place the radio into an idle state.
RAIL_RadioState_t	RAIL_GetRadioState (RAIL_Handle_t railHandle) Get the current radio state.
RAIL_RadioStateDetail_t	RAIL_GetRadioStateDetail (RAIL_Handle_t railHandle) Get the detailed current radio state.
RAIL_Status_t	RAIL_EnableCacheSynthCal (RAIL_Handle_t railHandle, bool enable) Enable/disable caching of synth calibration value.

Macros

#define	RAIL_TRANSITION_TIME_KEEP ((RAIL_TransitionTime_t) -1) A value to use in RAIL_StateTiming_t fields when calling RAIL_SetStateTiming() to keep that timing parameter at its current setting.
#define	RAIL_RF_STATE_DETAIL_INACTIVE (0U) Radio is inactive.
#define	RAIL_RF_STATE_DETAIL_IDLE_STATE (1U << RAIL_RF_STATE_DETAIL_IDLE_STATE_SHIFT) Radio is in or headed to the idle state.
#define	RAIL_RF_STATE_DETAIL_RX_STATE (1U << RAIL_RF_STATE_DETAIL_RX_STATE_SHIFT) Radio is in or headed to the receive state.
#define	RAIL_RF_STATE_DETAIL_TX_STATE (1U << RAIL_RF_STATE_DETAIL_TX_STATE_SHIFT) Radio is in or headed to the transmit state.
#define	RAIL_RF_STATE_DETAIL_TRANSITION (1U << RAIL_RF_STATE_DETAIL_TRANSITION_SHIFT) Radio is headed to the idle, receive, or transmit state.
#define	RAIL_RF_STATE_DETAIL_ACTIVE (1U << RAIL_RF_STATE_DETAIL_ACTIVE_SHIFT) Radio is actively transmitting or receiving.
#define	RAIL_RF_STATE_DETAIL_NO_FRAMES (1U << RAIL_RF_STATE_DETAIL_NO_FRAMES_SHIFT) Radio has frame detect disabled.
#define	RAIL_RF_STATE_DETAIL_LBT (1U << RAIL_RF_STATE_DETAIL_LBT_SHIFT) LBT/CSMA operation is currently ongoing.

```
#define RAIL_RF_STATE_DETAIL_CORE_STATE_MASK undefined
Mask for core radio state bits.
```

Enumeration Documentation

RAIL_RadioState_t

RAIL_RadioState_t

The state of the radio.

Enumerator	
RAIL_RF_STATE_INACTIVE	Radio is inactive.
RAIL_RF_STATE_ACTIVE	Radio is either idle or, in combination with the RX and TX states, receiving or transmitting a frame.
RAIL_RF_STATE_RX	Radio is in receive.
RAIL_RF_STATE_TX	Radio is in transmit.
RAIL_RF_STATE_IDLE	Radio is idle.
RAIL_RF_STATE_RX_ACTIVE	Radio is actively receiving a frame.
RAIL_RF_STATE_TX_ACTIVE	Radio is actively transmitting a frame.

Definition at line 2725 of file common/rail_types.h

RAIL_RadioStateDetail_t

RAIL_RadioStateDetail_t

The detailed state of the radio.

The three radio state bits [RAIL_RF_STATE_DETAIL_IDLE_STATE](#), [RAIL_RF_STATE_DETAIL_RX_STATE](#), and [RAIL_RF_STATE_DETAIL_TX_STATE](#) comprise a set of mutually exclusive core radio states. Only one (or none) of these bits can be set at a time. Otherwise, the value is invalid.

The precise meaning of each of these three core bits, when set, depends on the value of the two bits [RAIL_RF_STATE_DETAIL_TRANSITION](#) and [RAIL_RF_STATE_DETAIL_ACTIVE](#). When [RAIL_RF_STATE_DETAIL_TRANSITION](#) is set, the radio is transitioning into the core radio state corresponding to the set state bit. When it is clear, the radio is already in the core radio state that corresponds to the set state bit. When [RAIL_RF_STATE_DETAIL_ACTIVE](#) is set, the radio is actively transmitting or receiving. When it is clear, the radio is not actively transmitting or receiving. This bit will always be clear when [RAIL_RF_STATE_DETAIL_IDLE_STATE](#) is set, and will always be set when [RAIL_RF_STATE_DETAIL_TX_STATE](#) is set. Otherwise, the value is invalid.

The bit [RAIL_RF_STATE_DETAIL_NO_FRAMES](#) is set if the radio is currently operating with frame detection disabled, and clear otherwise. The bit [RAIL_RF_STATE_DETAIL_LBT_SHIFT](#) is set if an LBT/CSMA operation (e.g., performing CCA) is currently ongoing, and clear otherwise.

Enumerator	
RAIL_RF_STATE_DETAIL_IDLE_STATE_SHIFT	Shift position of RAIL_RF_STATE_DETAIL_IDLE_STATE bit.
RAIL_RF_STATE_DETAIL_RX_STATE_SHIFT	Shift position of RAIL_RF_STATE_DETAIL_RX_STATE bit.
RAIL_RF_STATE_DETAIL_TX_STATE_SHIFT	Shift position of RAIL_RF_STATE_DETAIL_TX_STATE bit.
RAIL_RF_STATE_DETAIL_TRANSITION_SHIFT	Shift position of RAIL_RF_STATE_DETAIL_TRANSITION bit.
RAIL_RF_STATE_DETAIL_ACTIVE_SHIFT	Shift position of RAIL_RF_STATE_DETAIL_ACTIVE bit.

RAIL_RF_STATE_DETAIL_NO_FRAMES_SHIFT	Shift position of RAIL_RF_STATE_DETAIL_NO_FRAMES bit.
RAIL_RF_STATE_DETAIL_LBT_SHIFT	Shift position of RAIL_RF_STATE_DETAIL_LBT bit.

Definition at line 2858 of file `common/rail_types.h`

RAIL_IdleMode_t

RAIL_IdleMode_t

An enumeration for the different types of supported idle modes.
These vary how quickly and destructively they put the radio into idle.

Enumerator	
RAIL_IDLE	Idle the radio by turning off receive and canceling any future scheduled receive or transmit operations.
RAIL_IDLE_ABORT	Idle the radio by turning off receive and any scheduled events.
RAIL_IDLE_FORCE_SHUTDOWN	Force the radio into a shutdown mode by stopping whatever state is in progress.
RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS	Similar to the RAIL_IDLE_FORCE_SHUTDOWN command, however, it will also clear any pending RAIL events related to receive and transmit.

Definition at line 2901 of file `common/rail_types.h`

Typedef Documentation

RAIL_TransitionTime_t

RAIL_TransitionTime_t

Suitable type for the supported transition time range.
Refer to platform-specific [RAIL_MINIMUM_TRANSITION_US](#) and [RAIL_MAXIMUM_TRANSITION_US](#) for the valid range of this type.

Definition at line 2672 of file `common/rail_types.h`

Function Documentation

RAIL_SetRxTransitions

RAIL_Status_t RAIL_SetRxTransitions ([RAIL_Handle_t](#) railHandle, const [RAIL_StateTransitions_t](#) * transitions)

Configure RAIL automatic state transitions after RX.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	transitions	A pointer to the state transitions to apply after reception.

Returns

Status code indicating success of the function call.

This function fails if unsupported transitions are passed in or if the radio is currently in the RX state. Success can transition to TX, RX, or IDLE, while error can transition to RX or IDLE. The timings of state transitions from the RX state are not guaranteed when packets are longer than 16 seconds on-air.

Definition at line 2392 of file common/rail.h

RAIL_GetRxTransitions

RAIL_Status_t RAIL_GetRxTransitions (RAIL_Handle_t railHandle, RAIL_StateTransitions_t * transitions)

Get the current RAIL automatic state transitions after RX.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	transitions	A pointer to the state transitions that RAIL applies after receive.

Returns

- Status code indicating a success of the function call.

Retrieves the current state transitions after RX and stores them in the transitions argument.

Definition at line 2405 of file common/rail.h

RAIL_SetTxTransitions

RAIL_Status_t RAIL_SetTxTransitions (RAIL_Handle_t railHandle, const RAIL_StateTransitions_t * transitions)

Configure RAIL automatic state transitions after TX.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	transitions	A pointer to the state transitions to apply after transmission.

Returns

- Status code indicating a success of the function call.

This function fails if unsupported transitions are passed in or if the radio is currently in the TX state. Success and error can each transition to RX or IDLE only, not TX. For the ability to run repeated transmits, see [RAIL_SetNextTxRepeat\(\)](#). Calling this function will clear any repeated transmissions set up by [RAIL_SetNextTxRepeat\(\)](#).

Definition at line 2421 of file common/rail.h

RAIL_GetTxTransitions

RAIL_Status_t RAIL_GetTxTransitions (RAIL_Handle_t railHandle, RAIL_StateTransitions_t * transitions)

Get the current RAIL automatic state transitions after TX.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	transitions	A pointer to the state transitions that RAIL applies after transmission.

Returns

- Status code indicating a success of the function call.

Retrieves the current state transitions after TX and stores them in the transitions argument.

Definition at line 2434 of file common/rail.h

RAIL_SetNextTxRepeat

```
RAIL_Status_t RAIL_SetNextTxRepeat (RAIL_Handle_t railHandle, const RAIL_TxRepeatConfig_t * repeatConfig)
```

Set up automatic repeated transmits after the next transmit.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	repeatConfig	A pointer to the configuration structure for repeated transmits.

Returns

- Status code indicating a success of the function call.

Repeated transmits will occur after an application-initiated transmit caused by calling one of the [Packet Transmit](#) APIs. The repetition will only occur after the first application-initiated transmit after this function is called. Future repeated transmits must be requested by calling this function again.

Each repeated transmit that occurs will have full [PTI Packet Trace](#) information, and will receive events such as [RAIL_EVENT_TX_PACKET_SENT](#) as normal.

If a TX error occurs during the repetition, the process will abort and the TX error transition from [RAIL_SetTxTransitions\(\)](#) will be used. If the repetition completes successfully, then the TX success transition from [RAIL_SetTxTransitions\(\)](#) will be used.

Use [RAIL_GetTxPacketsRemaining\(\)](#) if need to know how many transmit completion events are expected before the repeating sequence is done, or how many were not performed due to a transmit error.

Any call to [RAIL_Idle\(\)](#), [RAIL_StopTx\(\)](#), or [RAIL_SetTxTransitions\(\)](#) will clear the pending repeated transmits. The state will also be cleared by another call to this function. A DMP switch will clear this state only if the initial transmit triggering the repeated transmits has started.

One can change the repeated transmit configuration by re-calling this function with new parameters as long as that occurs prior to calling a [Packet Transmit](#) API. Passing a [RAIL_TxRepeatConfig_t::iterations](#) count of 0 will prevent the next transmit from repeating.

The application is responsible for populating the transmit data to be used by the repeated transmits via [RAIL_SetTxFifo\(\)](#) or [RAIL_WriteTxFifo\(\)](#). Data will be transmitted from the transmit FIFO. If the transmit FIFO does not have sufficient data to transmit, a TX error will be caused and a [RAIL_EVENT_TX_UNDERFLOW](#) will occur. In order to avoid an underflow, the application should queue data to be transmitted as early as possible. Consider using [RAIL_TX_OPTION_RESEND](#) if the same packet data is to be repeated: then the transmit FIFO only needs to be set/written once.

Do not call this function after starting a transmit operation via a [Packet Transmit](#) API call or before processing the final transmit completion event of a prior transmit. This function will fail to (re)configure the repetition if a transmit of any kind is ongoing, including during the time between an initial transmit and the end of a previously-configured repetition.

Note

- Use the compile time symbol [RAIL_SUPPORTS_TX_TO_TX](#) or the runtime call [RAIL_SupportsTxToTx\(\)](#) to check whether the platform supports this feature.

Definition at line 2495 of file common/rail.h

RAIL_GetTxPacketsRemaining

uint16_t RAIL_GetTxPacketsRemaining (RAIL_Handle_t railHandle)

Get the number of transmits remaining in a repeat operation.

Parameters

[in]	railHandle	A RAIL instance handle.

Must only be called from within event callback context when handling one of the [RAIL_EVENTS_TX_COMPLETION](#) events.

Returns

- transmits remaining as described below.

If the TX completion event is [RAIL_EVENT_TX_PACKET_SENT](#) the returned value indicates how many more such events are expected before the repeat transmit operation is done. Due to interrupt latency and timing, this may be an overcount if greater than 0 but is guaranteed to be accurate when 0.

If the TX completion event is an error, the returned value indicates the number of requested transmits that were not performed. For [RAIL_EVENT_TX_ABORTED](#) and [RAIL_EVENT_TX_UNDERFLOW](#) the count does not include the failing transmit itself. For the other errors where a transmit never started or was blocked, the count would include the failing transmit, which may be one higher than the configured [RAIL_TxRepeatConfig_t::iterations](#) if it was the original transmit that was blocked.

If an infinite repeat was configured, this will return [RAIL_TX_REPEAT_INFINITE_ITERATIONS](#).

Definition at line 2524 of file common/rail.h

RAIL_SetStateTiming

RAIL_Status_t RAIL_SetStateTiming (RAIL_Handle_t railHandle, RAIL_StateTiming_t * timings)

Configure RAIL automatic state transition timing.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	timings	A pointer to the timings used to configure the RAIL state machine. This structure is updated with the actual times that were set, if an input timing is invalid.

Returns

- Status code indicating a success of the function call.

The timings given are close to the actual transition time. However, a still uncharacterized software overhead occurs. Also, timings are not always adhered to when using an automatic transition after an error, due to the cleanup required to recover from the error.

Definition at line 2540 of file common/rail.h

RAIL_Idle

RAIL_Status_t RAIL_Idle (RAIL_Handle_t railHandle, RAIL_IdleMode_t mode, bool wait)

Place the radio into an idle state.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mode	The method for shutting down the radio.
[in]	wait	Whether this function should wait for the radio to reach idle before returning.

Returns

- [RAIL_STATUS_NO_ERROR](#) if the radio has idled by the time this function returns, or [RAIL_STATUS_SUSPENDED](#) if it is still in the process of idling.

This function is used to remove the radio from TX and RX states. How these states are left is defined by the mode parameter.

In multiprotocol, this API will also cause the radio to be yielded so that other tasks can be run. See [Yielding the Radio](#) for more details.

Definition at line 2560 of file `common/rail.h`

RAIL_GetRadioState

RAIL_RadioState_t RAIL_GetRadioState ([RAIL_Handle_t](#) railHandle)

Get the current radio state.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- An enumeration for the current radio state.

Returns the state of the radio as a bitmask containing: [RAIL_RF_STATE_IDLE](#), [RAIL_RF_STATE_RX](#), [RAIL_RF_STATE_TX](#), and [RAIL_RF_STATE_ACTIVE](#). [RAIL_RF_STATE_IDLE](#), [RAIL_RF_STATE_RX](#), and [RAIL_RF_STATE_TX](#) bits are mutually exclusive. The radio can transition through intermediate states, which are not reported but are instead considered part of the state most closely associated. For example, when the radio is warming up or shutting down the transmitter or receiver, this function returns [RAIL_RF_STATE_TX](#) or [RAIL_RF_STATE_RX](#), respectively. When transitioning directly from RX to TX or vice-versa, this function returns the earlier state.

Note

- For a more detailed radio state, see [RAIL_GetRadioStateDetail\(\)](#).

Definition at line 2584 of file `common/rail.h`

RAIL_GetRadioStateDetail

RAIL_RadioStateDetail_t RAIL_GetRadioStateDetail ([RAIL_Handle_t](#) railHandle)

Get the detailed current radio state.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- An enumeration for the current detailed radio state.

Returns the state of the radio as a bitmask. The three core radio states IDLE, RX, and TX are represented by mutually exclusive bits [RAIL_RF_STATE_DETAIL_IDLE_STATE](#), [RAIL_RF_STATE_DETAIL_RX_STATE](#), and [RAIL_RF_STATE_DETAIL_TX_STATE](#) respectively. If the radio is transitioning between these three states, the returned bitmask will have [RAIL_RF_STATE_DETAIL_TRANSITION](#) set along with a bit corresponding to the destination core radio state. If, while in the receive state, the radio is actively receiving a packet, [RAIL_RF_STATE_DETAIL_ACTIVE](#) will be set; otherwise, this bit will be clear. If frame detection is disabled, [RAIL_RF_STATE_DETAIL_NO_FRAMES](#) in the returned state bitmask will be set; otherwise, this bit will be clear. If the radio is performing an LBT/CSMA operation (e.g., a backoff period) [RAIL_RF_STATE_DETAIL_LBT](#) in the returned state bitmask will be set; otherwise, this bit will be clear.

For the most part, the more detailed radio states returned by this API correspond to radio states returned by [RAIL_GetRadioState\(\)](#) as follows:

```
RAIL_RadioStateDetail_tRAIL_RadioState_t RAIL_RF_STATE_DETAIL_INACTIVE RAIL_RF_STATE_INACTIVE
RAIL_RF_STATE_DETAIL_IDLE_STATE | RAIL_STATE_DETAIL_TRANSITION If RX overflow or leaving RX unforced:
RAIL_RF_STATE_RX Else if leaving TX unforced: RAIL_RF_STATE_TX Else: RAIL_RF_STATE_IDLE
RAIL_RF_STATE_DETAIL_IDLE_STATE RAIL_RF_STATE_IDLE RAIL_RF_STATE_DETAIL_IDLE_STATE | RAIL_STATE_DETAIL_LBT
RAIL_RF_STATE_TX RAIL_RF_STATE_DETAIL_RX_STATE | RAIL_STATE_DETAIL_TRANSITION If leaving TX: RAIL_RF_STATE_TX
Else: RAIL_RF_STATE_RX RAIL_RF_STATE_DETAIL_RX_STATE | RAIL_RF_STATE_DETAIL_TRANSITION |
RAIL_RF_STATE_DETAIL_NO_FRAMES If leaving TX: RAIL_RF_STATE_TX Else: RAIL_RF_STATE_RX
RAIL_RF_STATE_DETAIL_RX_STATE RAIL_RF_STATE_RX RAIL_RF_STATE_DETAIL_RX_STATE |
RAIL_RF_STATE_DETAIL_NO_FRAMES RAIL_RF_STATE_RX RAIL_RF_STATE_DETAIL_RX_STATE | RAIL_RF_STATE_DETAIL_LBT
RAIL_RF_STATE_RX RAIL_RF_STATE_DETAIL_RX_STATE | RAIL_RF_STATE_DETAIL_NO_FRAMES |
RAIL_RF_STATE_DETAIL_LBT RAIL_RF_STATE_RX RAIL_RF_STATE_DETAIL_RX_STATE | RAIL_RF_STATE_DETAIL_ACTIVE
RAIL_RF_STATE_RX_ACTIVE RAIL_RF_STATE_DETAIL_TX_STATE | RAIL_RF_STATE_TRANSITION If leaving RX:
RAIL_RF_STATE_RX Else: RAIL_RF_STATE_TX RAIL_RF_STATE_DETAIL_TX_STATE | RAIL_RF_STATE_ACTIVE
RAIL_RF_STATE_TX_ACTIVE
```

Definition at line 2650 of file `common/rail.h`

RAIL_EnableCacheSynthCal

RAIL_Status_t RAIL_EnableCacheSynthCal ([RAIL_Handle_t](#) railHandle, bool enable)

Enable/disable caching of synth calibration value.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	A boolean to enable or disable caching of synth calibration.

Returns

- Status code indicating success of the function call.

Once enabled, the sequencer will start caching synth calibration values for channels and apply them instead of performing calibration on every state transition and channel change. This will increase the transition time for the first time calibration is performed. Subsequent state transitions will be faster. The cache size is 2. [RAIL_IEEE802154_SUPPORTS_RX_CHANNEL_SWITCHING](#) internally uses this feature and there is no need to enable/disable it. This function returns [RAIL_STATUS_INVALID_STATE](#) if we try to disable it while [RAIL_IEEE802154_SUPPORTS_RX_CHANNEL_SWITCHING](#) is enabled.

Note

- This function will improve the minimum timings that can be achieved in [RAIL_StateTiming_t::idleToRx](#), [RAIL_StateTiming_t::idleToTx](#), [RAIL_StateTiming_t::rxToTx](#), [RAIL_StateTiming_t::txToRx](#) and [RAIL_StateTiming_t::txToTx](#). A call to [RAIL_SetStateTiming\(\)](#) is needed to achieve lower transition times.

On a protocol switch the cache is cleared, so it is not suitable for applications where a protocol switch happens frequently, like with Dynamic Multiprotocol.

Definition at line 2679 of file `common/rail.h`

RAIL_StateTiming_t

A timing configuration structure for the RAIL State Machine.

Configure the timings of the radio state transitions for common situations. All of the listed timings are in microseconds. Transitions from an active radio state to idle are not configurable, and will always happen as fast as possible. No timing value can exceed platform-specific `RAIL_MAXIMUM_TRANSITION_US`. Use `RAIL_TRANSITION_TIME_KEEP` to keep an existing setting.

For `idleToRx`, `idleToTx`, `rxToTx`, `txToRx`, and `txToTx` a value of 0 for the transition time means that the specified transition should happen as fast as possible, even if the timing cannot be as consistent. Otherwise, the timing value cannot be below the platform-specific `RAIL_MINIMUM_TRANSITION_US`.

For `idleToTx`, `rxToTx`, and `txToTx` setting a longer `RAIL_TxPowerConfig_t::rampTime` may result in a larger minimum value.

For `rxSearchTimeout` and `txToRxSearchTimeout`, there is no minimum value. A value of 0 disables the feature, functioning as an infinite timeout.

Public Attributes

<code>RAIL_TransitionTime_t</code>	<code>idleToRx</code> Transition time from IDLE to RX.
<code>RAIL_TransitionTime_t</code>	<code>txToRx</code> Transition time from TX to RX.
<code>RAIL_TransitionTime_t</code>	<code>idleToTx</code> Transition time from IDLE to TX.
<code>RAIL_TransitionTime_t</code>	<code>rxToTx</code> Transition time from RX packet to TX.
<code>RAIL_TransitionTime_t</code>	<code>rxSearchTimeout</code> Length of time the radio will search for a packet when coming from idle or RX.
<code>RAIL_TransitionTime_t</code>	<code>txToRxSearchTimeout</code> Length of time the radio will search for a packet when coming from TX.
<code>RAIL_TransitionTime_t</code>	<code>txToTx</code> Transition time from TX packet to TX.

Public Attribute Documentation

idleToRx

`RAIL_TransitionTime_t RAIL_StateTiming_t::idleToRx`

Transition time from IDLE to RX.

Definition at line 2706 of file `common/rail_types.h`

txToRx


```
RAIL_TransitionTime_t RAIL_StateTiming_t::txToRx
```

Transition time from TX to RX.

Definition at line 2708 of file `common/rail_types.h`

idleToTx

```
RAIL_TransitionTime_t RAIL_StateTiming_t::idleToTx
```

Transition time from IDLE to TX.

Definition at line 2710 of file `common/rail_types.h`

rxToTx

```
RAIL_TransitionTime_t RAIL_StateTiming_t::rxToTx
```

Transition time from RX packet to TX.

Definition at line 2712 of file `common/rail_types.h`

rxSearchTimeout

```
RAIL_TransitionTime_t RAIL_StateTiming_t::rxSearchTimeout
```

Length of time the radio will search for a packet when coming from idle or RX.

Definition at line 2714 of file `common/rail_types.h`

txToRxSearchTimeout

```
RAIL_TransitionTime_t RAIL_StateTiming_t::txToRxSearchTimeout
```

Length of time the radio will search for a packet when coming from TX.

Definition at line 2716 of file `common/rail_types.h`

txToTx

```
RAIL_TransitionTime_t RAIL_StateTiming_t::txToTx
```

Transition time from TX packet to TX.

Definition at line 2718 of file `common/rail_types.h`

RAIL_StateTransitions_t

Used to specify radio states to transition to on success or failure.

Public Attributes

- RAIL_RadioState_
t

success

Indicate the state the radio should return to after a successful action.
- RAIL_RadioState_
t

error

Indicate the state the radio should return to after an error.

Public Attribute Documentation

success

RAIL_RadioState_t RAIL_StateTransitions_t::success

Indicate the state the radio should return to after a successful action.

Definition at line 2825 of file common/rail_types.h

error

RAIL_RadioState_t RAIL_StateTransitions_t::error

Indicate the state the radio should return to after an error.

Definition at line 2829 of file common/rail_types.h

EFR32xG2x

EFR32xG2x

EFR32xG2x-specific State Transitions.

Macros

```
#define RAIL_MINIMUM_TRANSITION_US (100U)
    The minimum value for a consistent RAIL transition.

#define RAIL_MAXIMUM_TRANSITION_US (1000000U)
    The maximum value for a consistent RAIL transition.
```

Slxx3xx

Slxx3xx

Slxx3xx-specific State Transitions.

Macros

#define	RAIL_MINIMUM_TRANSITION_US (75U) The minimum value for a consistent RAIL transition.
#define	RAIL_MAXIMUM_TRANSITION_US (1000000U) The maximum value for a consistent RAIL transition.

System Timing

System Timing

Functionality related to the RAIL timer and general system time.

These functions can be used to get information about the current system time or to manipulate the RAIL timer.

The system time returned by [RAIL_GetTime\(\)](#) is in the same timebase that is used throughout RAIL. Any callbacks or structures that provide a timestamp, such as [RAIL_RxPacketDetails_t::timeReceived](#), will use the same timebase as will any APIs that accept an absolute time for scheduling their action. Throughout the documentation, the timebase is referred to as the RAIL timebase. The timebase is currently a value in microseconds from [RAIL_Init\(\)](#) time, which means that it will wrap every 1.19 hours. $(2^{32} - 1) / (3600 \text{ sec/hr} * 1000000 \text{ us/sec})$.

The provided timer is hardware-backed and interrupt-driven. It can be used for timing any event in the system, but is especially helpful for timing protocol-based state machines and other systems that interact with the radio. To avoid processing the expiration in interrupt context, leave the cb parameter passed to [RAIL_SetTimer\(\)](#) as NULL and poll for expiration with the [RAIL_IsTimerExpired\(\)](#) function. See below for an example of the interrupt driven method of interacting with the timer.

```
void timerCb(RAIL_Handle_t railHandle)
{
    // Timer callback action
}

void main(void)
{
    // Initialize RAIL ...

    // Set up a timer for 1 ms from now
    RAIL_SetTimer(railHandle, 1000, RAIL_TIME_RELATIVE, &timerCb);

    // Run main loop
    while(1);
}
```

If multiple software timers are needed to be run off of the one available hardware timer, enable a software multiplexing layer within RAIL using the [RAIL_ConfigMultiTimer\(\)](#) function. This will allow you to set up as many timers as you want using the [RAIL_*MultiTimer\(\)](#) functions. See the example below for using the multitimer functionality.

```
// Declare timer structures in global space or somewhere that will exist
// until the callback has fired
RAIL_MultiTimer_t tmr1, tmr2;

void timerCb(RAIL_MultiTimer_t *tmr,
             RAIL_Time_t expectedTimeOfEvent,
             void *cbArg)
{
    if (tmr == &tmr1) {
        // Timer 1 action
    } else {
        // Timer 2 action
    }
}

void main(void)
{
    // Initialize RAIL ...

    RAIL_ConfigMultiTimer(true);

    // Set up one timer for 1 ms from now and one at time 2000000 in the RAIL
    // timebase
    RAIL_SetMultiTimer(&tmr1, 1000, RAIL_TIME_RELATIVE, &timerCb, NULL);
    RAIL_SetMultiTimer(&tmr2, 2000000, RAIL_TIME_ABSOLUTE, &timerCb, NULL);

    // Run main loop
    while(1);
}
```

Modules

[RAIL_MultiTimer_t](#)

[RAIL_PacketTimeStamp_t](#)

[RAIL_MultiTimer](#)

Enumerations

```
enum RAIL_TimeMode_t {
    RAIL_TIME_ABSOLUTE = 0
    RAIL_TIME_DELAY = 1
    RAIL_TIME_DISABLED = 2
}
Specify a time offset in RAIL APIs.
```

```
enum RAIL_PacketTimePosition_t {
    RAIL_PACKET_TIME_INVALID = 0
    RAIL_PACKET_TIME_DEFAULT = 1
    RAIL_PACKET_TIME_AT_PREAMBLE_START = 2
    RAIL_PACKET_TIME_AT_PREAMBLE_START_USED_TOTAL = 3
    RAIL_PACKET_TIME_AT_SYNC_END = 4
    RAIL_PACKET_TIME_AT_SYNC_END_USED_TOTAL = 5
    RAIL_PACKET_TIME_AT_PACKET_END = 6
    RAIL_PACKET_TIME_AT_PACKET_END_USED_TOTAL = 7
    RAIL_PACKET_TIME_COUNT
}
The available packet timestamp position choices.
```

Typedefs

typedef uint32_t	RAIL_Time_t Time in microseconds.
typedef void(*)	RAIL_TimerCallback_t (RAIL_Handle_t railHandle) A pointer to the callback called when the RAIL timer expires.
typedef void(*)	RAIL_MultiTimerCallback_t (struct RAIL_MultiTimer *tmr, RAIL_Time_t expectedTimeOfEvent, void *cbArg) Callback fired when timer expires.

Functions

RAIL_Time_t	RAIL_GetTime (void) Get the current RAIL time.
RAIL_Status_t	RAIL_SetTime (RAIL_Time_t time) Set the current RAIL time.
RAIL_Status_t	RAIL_DelayUs (RAIL_Time_t microseconds) Blocking delay routine for a specified number of microseconds.
RAIL_Status_t	RAIL_SetTimer (RAIL_Handle_t railHandle, RAIL_Time_t time, RAIL_TimeMode_t mode, RAIL_TimerCallback_t cb) Schedule a timer to expire using the RAIL timebase.
RAIL_Time_t	RAIL_GetTimer (RAIL_Handle_t railHandle) Return the absolute time that the RAIL timer was configured to expire.
RAIL_Status_t	RAIL_CancelTimer (RAIL_Handle_t railHandle) Stop the currently scheduled RAIL timer.
bool	RAIL_IsTimerExpired (RAIL_Handle_t railHandle) Check whether the RAIL timer has expired.
bool	RAIL_IsTimerRunning (RAIL_Handle_t railHandle) Check whether the RAIL timer is currently running.
bool	RAIL_ConfigMultiTimer (bool enable) Configure the RAIL software timer feature.
RAIL_Status_t	RAIL_SetMultiTimer (RAIL_MultiTimer_t *tmr, RAIL_Time_t expirationTime, RAIL_TimeMode_t expirationMode, RAIL_MultiTimerCallback_t callback, void *cbArg) Start a multimer instance.
bool	RAIL_CancelMultiTimer (RAIL_MultiTimer_t *tmr) Stop the currently scheduled RAIL multimer.
bool	RAIL_IsMultiTimerRunning (RAIL_MultiTimer_t *tmr) Check if a given timer is running.
bool	RAIL_IsMultiTimerExpired (RAIL_MultiTimer_t *tmr) Check if a given timer has expired.
RAIL_Time_t	RAIL_GetMultiTimer (RAIL_MultiTimer_t *tmr, RAIL_TimeMode_t timeMode) Get time left before a given timer instance expires.

Enumeration Documentation

RAIL_TimeMode_t

RAIL_TimeMode_t

Specify a time offset in RAIL APIs.

Different APIs use the same constants and may provide more specifics about how they're used but the general use for each is described below.

Enumerator

RAIL_TIME_ABSOLUTE	The time specified is an exact time in the RAIL timebase.
RAIL_TIME_DELAY	The time specified is relative to the current time.
RAIL_TIME_DISABLED	The specified time is invalid and should be ignored.

Definition at line 225 of file common/rail_types.h

RAIL_PacketTimePosition_t

RAIL_PacketTimePosition_t

The available packet timestamp position choices.

Enumerator

RAIL_PACKET_TIME_INVALID	Indicate that a timestamp is not to be or was not provided.
RAIL_PACKET_TIME_DEFAULT	Request the choice most expedient for RAIL to calculate, which may depend on the radio and/or its configuration.
RAIL_PACKET_TIME_AT_PREAMBLE_START	Request the timestamp corresponding to the first preamble bit sent or received.
RAIL_PACKET_TIME_AT_PREAMBLE_START_USED_TOTAL	Request the timestamp corresponding to the first preamble bit sent or received.
RAIL_PACKET_TIME_AT_SYNC_END	Request the timestamp corresponding to right after its last SYNC word bit has been sent or received.
RAIL_PACKET_TIME_AT_SYNC_END_USED_TOTAL	Request the timestamp corresponding to right after its last SYNC word bit has been sent or received.
RAIL_PACKET_TIME_AT_PACKET_END	Request the timestamp corresponding to right after its last bit has been sent or received.
RAIL_PACKET_TIME_AT_PACKET_END_USED_TOTAL	Request the timestamp corresponding to right after its last bit has been sent or received.
RAIL_PACKET_TIME_COUNT	A count of the choices in this enumeration.

Definition at line 312 of file common/rail_types.h

Typedef Documentation

RAIL_Time_t

RAIL_Time_t

Time in microseconds.

Definition at line 208 of file common/rail_types.h

RAIL_TimerCallback_t

```
typedef void(* RAIL_TimerCallback_t) (RAIL_Handle_t railHandle) )(RAIL_Handle_t railHandle)
```

A pointer to the callback called when the RAIL timer expires.

Parameters

[in]	railHandle	The RAIL handle that was used in the RAIL_SetTimer() call.

Definition at line 216 of file common/rail_types.h

RAIL_MultiTimerCallback_t

```
RAIL_MultiTimerCallback_t )(struct RAIL_MultiTimer *tmr, RAIL_Time_t expectedTimeOfEvent, void *cbArg)
```

Callback fired when timer expires.

Parameters

[in]	tmr	A pointer to an expired timer.
[in]	expectedTimeOfEvent	An absolute time event fired.
[in]	cbArg	A user-supplied callback argument. Since this callback doesn't include a parameter for the active RAIL_Handle_t that called RAIL_SetMultiTimer() it might be handy to pass that RAIL handle as the cbArg when calling RAIL_SetMultiTimer() .

Definition at line 278 of file common/rail_types.h

Function Documentation

RAIL_GetTime

```
RAIL_Time_t RAIL_GetTime (void )
```

Get the current RAIL time.

Parameters

N/A		

Returns

- The RAIL time in microseconds. Note that this wraps after about 1.19 hours since it's stored in a 32 bit value.
- Returns the current time in the RAIL timebase (microseconds). It can be used to compare with packet timestamps or to schedule transmits.

Definition at line 1062 of file common/rail.h

RAIL_SetTime

```
RAIL_Status_t RAIL_SetTime (RAIL\_Time\_t time)
```

Set the current RAIL time.

Parameters

[in]	time	Set the RAIL timebase to this value in microseconds.

Returns

- Status code indicating the success of the function call.

Warnings

- Use this API only for testing purposes or in very limited circumstances during RAIL Timer Synchronization. Undefined behavior can result by calling it in multiprotocol or when the radio is not idle or timed events are active. Applications using [RAIL_GetTime\(\)](#) may not be designed for discontinuous changes to the RAIL time base.

Definition at line 1077 of file common/rail.h

RAIL_DelayUs

RAIL_Status_t RAIL_DelayUs ([RAIL_Time_t](#) microseconds)

Blocking delay routine for a specified number of microseconds.

Parameters

[in]	microseconds	Delay duration in microseconds.

Returns

- Status code indicating success of the function call.

Use this RAIL API only for short blocking delays because it has less overhead than calling [RAIL_GetTime\(\)](#) in a loop.

Note

- Passing large delay values may give unpredictable results or trigger the Watchdog reset.
Also, this function will start the clocks required for the RAIL timebase if they are not running, except between [RAIL_Sleep\(\)](#) and [RAIL_Wake\(\)](#) where the timer must remain stopped.
Interrupts are not disabled during the delay, so the delay may be longer if an interrupt extends beyond the delay duration.

Definition at line 1096 of file common/rail.h

RAIL_SetTimer

RAIL_Status_t RAIL_SetTimer ([RAIL_Handle_t](#) railHandle, [RAIL_Time_t](#) time, [RAIL_TimeMode_t](#) mode, [RAIL_TimerCallback_t](#) cb)

Schedule a timer to expire using the RAIL timebase.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	time	The timer's expiration time in the RAIL timebase.
[in]	mode	Indicates whether the time argument is an absolute RAIL time or relative to the current RAIL time. Specifying mode RAIL_TIME_DISABLED is the same as calling RAIL_CancelTimer() .
[in]	cb	A pointer to a callback function that RAIL will call when the timer expires. May be NULL if no callback is desired.

Returns

- [RAIL_STATUS_NO_ERROR](#) on success and [RAIL_STATUS_INVALID_PARAMETER](#) if the timer can't be scheduled.

Configures a timer to expire after a period in the RAIL timebase. This timer can be used to implement low-level protocol features.

Warnings

- Attempting to schedule the timer when it is still running from a previous request is bad practice, unless the cb callback is identical to that used in the previous request, in which case the timer is rescheduled to the new time. Note that if the original timer expires as it is being rescheduled, the callback may or may not occur. It is generally good practice to cancel a running timer before rescheduling it to minimize ambiguity.

Definition at line 1124 of file common/rail.h

RAIL_GetTimer

```
RAIL_Time_t RAIL_GetTimer (RAIL_Handle_t railHandle)
```

Return the absolute time that the RAIL timer was configured to expire.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The absolute time that this timer was set to expire.

Provides the absolute time regardless of the [RAIL_TimeMode_t](#) that was passed into [RAIL_SetTimer\(\)](#). Note that the time might be in the past if the timer has already expired. The return value is undefined if the timer was never set.

Definition at line 1140 of file common/rail.h

RAIL_CancelTimer

```
RAIL_Status_t RAIL_CancelTimer (RAIL_Handle_t railHandle)
```

Stop the currently scheduled RAIL timer.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- [RAIL_STATUS_NO_ERROR](#) on success and [RAIL_STATUS_INVALID_CALL](#) if the timer is not running.

Cancels the timer. If this function is called before the timer expires, the cb callback specified in the earlier [RAIL_SetTimer\(\)](#) call will never be called.

Definition at line 1153 of file common/rail.h

RAIL_IsTimerExpired

```
bool RAIL_IsTimerExpired (RAIL_Handle_t railHandle)
```

Check whether the RAIL timer has expired.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if the previously scheduled timer has expired and false otherwise.

Polling with this function is an alternative to the callback.

Definition at line 1164 of file common/rail.h

RAIL_IsTimerRunning

```
bool RAIL_IsTimerRunning (RAIL_Handle_t railHandle)
```

Check whether the RAIL timer is currently running.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if the timer is running and false if the timer has expired or was never set.

Definition at line 1173 of file common/rail.h

RAIL_ConfigMultiTimer

```
bool RAIL_ConfigMultiTimer (bool enable)
```

Configure the RAIL software timer feature.

Parameters

[in]	enable	Enables/disables the RAIL multimer.

Returns

- true if the multimer was successfully enabled/disabled, false otherwise.

Turning this on will add a software timer layer above the physical RAIL timer so that the user can have as many timers as desired. It is not necessary to call this function if the MultiTimer APIs are not used.

Note

- This function must be called before calling [RAIL_SetMultiTimer\(\)](#). This function is a no-op on multiprotocol as this layer is already used under the hood. Do not call this function while the RAIL timer is running. Call [RAIL_IsTimerRunning\(\)](#) before enabling/disabling the multimer. If the multimer is not needed, do not call this function to allow the multimer code to be dead stripped. If the multimer is enabled for use, the multimer and timer APIs can both be used. However, no timer can be in use while this function is being called.

Definition at line 1198 of file common/rail.h

RAIL_SetMultiTimer

```
RAIL_Status_t RAIL_SetMultiTimer (RAIL_MultiTimer_t * tmr, RAIL_Time_t expirationTime, RAIL_TimeMode_t expirationMode, RAIL_MultiTimerCallback_t callback, void * cbArg)
```

Start a multimer instance.

Parameters

[inout]	tmr	A pointer to the timer instance to start.
[in]	expirationTime	A time when the timer is set to expire.
[in]	expirationMode	Select mode of expirationTime. See RAIL_TimeMode_t .
[in]	callback	A function to call on timer expiry. See RAIL_MultiTimerCallback_t . May be NULL if no callback is desired.
[in]	cbArg	An extra callback function parameter for the user application. Since the RAIL_MultiTimerCallback_t callback function lacks a RAIL_Handle_t parameter this can be used to pass the current RAIL handle if desired.

Returns

- [RAIL_STATUS_NO_ERROR](#) on success.
- [RAIL_STATUS_INVALID_PARAMETER](#) if tmr has an illegal value or if timeout is in the past.

Note

- It is legal to start an already running timer. If this is done, the timer will first be stopped before the new configuration is applied. If expirationTime is 0, the callback is called immediately.

Definition at line 1222 of file common/rail.h

RAIL_CancelMultiTimer

```
bool RAIL_CancelMultiTimer (RAIL_MultiTimer_t * tmr)
```

Stop the currently scheduled RAIL multimer.

Parameters

[inout]	tmr	A pointer to a RAIL timer instance.

Returns

- true if the timer was successfully canceled; false if the timer was not running.
- Cancels the timer. If this function is called before the timer expires, the cb callback specified in the earlier [RAIL_SetTimer\(\)](#) call will never be called.

Definition at line 1239 of file common/rail.h

RAIL_IsMultiTimerRunning

```
bool RAIL_IsMultiTimerRunning (RAIL_MultiTimer_t * tmr)
```

Check if a given timer is running.

Parameters

[inout]	tmr	A pointer to the timer instance.

Returns

- true if the timer is running; false if the timer is not running or tmr is not a timer instance.

Definition at line 1248 of file common/rail.h

RAIL_IsMultiTimerExpired

```
bool RAIL_IsMultiTimerExpired (RAIL_MultiTimer_t * tmr)
```

Check if a given timer has expired.

Parameters

[inout]	tmr	A pointer to the timer instance.

Returns

- true if the timer has expired or tmr is not a timer instance; false if the timer is running.

Definition at line 1257 of file common/rail.h

RAIL_GetMultiTimer

```
RAIL_Time_t RAIL_GetMultiTimer (RAIL_MultiTimer_t * tmr, RAIL_TimeMode_t timeMode)
```

Get time left before a given timer instance expires.

Parameters

[inout]	tmr	A pointer to the timer instance to query.
[in]	timeMode	Indicates how the function provides the time remaining. By choosing RAIL_TimeMode_t::RAIL_TIME_ABSOLUTE , the function returns the absolute expiration time, and by choosing RAIL_TimeMode_t::RAIL_TIME_DELAY , the function returns the amount of time remaining before the timer's expiration.

Returns

- Time left expressed in RAIL's time units. 0 if the timer is not running or has already expired.

Definition at line 1273 of file common/rail.h

RAIL_MultiTimer_t

RAIL timer state structure.

This structure is filled out and maintained internally only. The user/application should not alter any elements of this structure.

Public Attributes

RAIL_Time_t	absOffset	Absolute time before the next event.
RAIL_Time_t	relPeriodic	Relative, periodic time between events; 0 = timer is oneshot.
RAIL_MultiTimerC callback_t	callback	A user callback.
void *	cbArg	A user callback argument.
struct RAIL_MultiTimer *	next	A pointer to the next soft timer structure.
uint8_t	priority	A priority of the callback; 0 = highest priority; 255 = lowest.
bool	isRunning	Indicates the timer is currently running.
bool	doCallback	Indicates the callback needs to run.

Public Attribute Documentation

absOffset

RAIL_Time_t RAIL_MultiTimer_t::absOffset

Absolute time before the next event.

Definition at line 291 of file common/rail_types.h

relPeriodic

RAIL_Time_t RAIL_MultiTimer_t::relPeriodic

Relative, periodic time between events; 0 = timer is oneshot.

Definition at line 293 of file common/rail_types.h

callback


```
RAIL_MultiTimerCallback_t RAIL_MultiTimer_t::callback
```

A user callback.

Definition at line 295 of file `common/rail_types.h`

cbArg

```
void* RAIL_MultiTimer_t::cbArg
```

A user callback argument.

Definition at line 297 of file `common/rail_types.h`

next

```
struct RAIL_MultiTimer* RAIL_MultiTimer_t::next
```

A pointer to the next soft timer structure.

Definition at line 299 of file `common/rail_types.h`

priority

```
uint8_t RAIL_MultiTimer_t::priority
```

A priority of the callback; 0 = highest priority; 255 = lowest.

Definition at line 301 of file `common/rail_types.h`

isRunning

```
bool RAIL_MultiTimer_t::isRunning
```

Indicates the timer is currently running.

Definition at line 303 of file `common/rail_types.h`

doCallback

```
bool RAIL_MultiTimer_t::doCallback
```

Indicates the callback needs to run.

Definition at line 305 of file `common/rail_types.h`

RAIL_PacketTimeStamp_t

Information for calculating and representing a packet timestamp.

Public Attributes

RAIL_Time_t	packetTime Timestamp of the packet in the RAIL timebase.
uint16_t	totalPacketBytes A value specifying the total length in bytes of the packet used when calculating the packetTime requested by the timePosition field.
RAIL_PacketTimePosition_t	timePosition A RAIL_PacketTimePosition_t value specifying the packet position to return in the packetTime field.
RAIL_Time_t	packetDurationUs In RX for EFR32xG25 only: A value specifying the on-air duration of the data packet, starting with the first bit of the PHR (i.e., end of sync word); preamble and sync word duration are hence excluded.

Public Attribute Documentation

packetTime

RAIL_Time_t RAIL_PacketTimeStamp_t::packetTime

Timestamp of the packet in the RAIL timebase.

Definition at line 390 of file common/rail_types.h

totalPacketBytes

uint16_t RAIL_PacketTimeStamp_t::totalPacketBytes

A value specifying the total length in bytes of the packet used when calculating the packetTime requested by the timePosition field.

This should account for all bytes sent over the air after the Preamble and Sync word(s) including CRC bytes.

Definition at line 397 of file common/rail_types.h

timePosition

RAIL_PacketTimePosition_t RAIL_PacketTimeStamp_t::timePosition

A [RAIL_PacketTimePosition_t](#) value specifying the packet position to return in the packetTime field.

If this is [RAIL_PACKET_TIME_DEFAULT](#), this field will be updated with the actual position corresponding to the packetTime value filled in by a call using this structure.

Definition at line 405 of file common/rail_types.h

packetDurationUs

RAIL_Time_t RAIL_PacketTimeStamp_t::packetDurationUs

In RX for EFR32xG25 only: A value specifying the on-air duration of the data packet, starting with the first bit of the PHR (i.e., end of sync word); preamble and sync word duration are hence excluded.

In Tx for all platforms: A value specifying the on-air duration of the data packet, starting at the preamble (i.e. includes preamble, sync word, PHR, payload and FCS). This value can be used to compute duty cycles. **Note**

- This field is currently valid only for normal transmits but not Auto-Ack transmits which set the field to zero.

Definition at line 419 of file common/rail_types.h

RAIL_MultiTimer

Forward structure declaration of [RAIL_MultiTimer_t](#).

TX Channel Hopping

TX Channel Hopping

Modules

[RAIL_TxChannelHoppingConfigEntry_t](#)

[RAIL_TxChannelHoppingConfig_t](#)

Macros

`#define` `RAIL_CHANNEL_HOPPING_BUFFER_SIZE_PER_CHANNEL_WORST_CASE` (65U)
The worst-case platform-agnostic static amount of memory needed per channel for channel hopping, measured in 32 bit words, regardless of the size of radio configuration structures.

RAIL_TxChannelHoppingConfigEntry_t

Structure that represents one of the channels that is part of a [RAIL_TxChannelHoppingConfig_t](#) sequence of channels used in channel hopping.

Public Attributes

uint16_t	channel	The channel number to be used for this entry in the channel hopping sequence.
uint8_t	reserved	Pad bytes reserved for future use and currently ignored.
uint32_t	delay	Idle time in microseconds to wait before transmitting on the channel indicated by this entry.

Public Attribute Documentation

channel

```
uint16_t RAIL_TxChannelHoppingConfigEntry_t::channel
```

The channel number to be used for this entry in the channel hopping sequence.

If this is an invalid channel for the current PHY, the call to [RAIL_SetNextTxRepeat\(\)](#) will fail.

Definition at line 2959 of file `common/rail_types.h`

reserved

```
uint8_t RAIL_TxChannelHoppingConfigEntry_t::reserved[2]
```

Pad bytes reserved for future use and currently ignored.

Definition at line 2963 of file `common/rail_types.h`

delay

```
uint32_t RAIL_TxChannelHoppingConfigEntry_t::delay
```

Idle time in microseconds to wait before transmitting on the channel indicated by this entry.

Definition at line 2968 of file `common/rail_types.h`

RAIL_TxChannelHoppingConfig_t

Wrapper struct that will contain the sequence of [RAIL_TxChannelHoppingConfigEntry_t](#) that represents the channel sequence to use during TX Channel Hopping.

Public Attributes

uint32_t *	buffer	Pointer to contiguous global read-write memory that will be used by RAIL to store channel hopping information throughout its operation.
uint16_t	bufferLength	This parameter must be set to the length of the buffer array, in 32 bit words.
uint8_t	numberOfChannels	The number of channels in the channel hopping sequence, which is the number of elements in the array that entries points to.
uint8_t	reserved	Pad byte reserved for future use and currently ignored.
RAIL_TxChannelHoppingConfigEntry_t *	entries	A pointer to the first element of an array of RAIL_TxChannelHoppingConfigEntry_t that represents the channels used during channel hopping.

Public Attribute Documentation

buffer

```
uint32_t* RAIL_TxChannelHoppingConfig_t::buffer
```

Pointer to contiguous global read-write memory that will be used by RAIL to store channel hopping information throughout its operation.

It need not be initialized and applications should never write data anywhere in this buffer.

Note

- The size of this buffer must be at least as large as $3 + \text{RAIL\_CHANNEL\_HOPPING\_BUFFER\_SIZE\_PER\_CHANNEL} * \text{numberOfChannels}$, plus the sum of the sizes of the radioConfigDeltaAdd's of the required channels, plus the size of the radioConfigDeltaSubtract. In the case that one channel appears two or more times in your channel sequence (e.g., 1, 2, 3, 2), you must account for the radio configuration size that number of times (i.e., need to count channel 2's radio configuration size twice for the given example). The buffer is for internal use to the library.

Definition at line 2995 of file `common/rail_types.h`

bufferLength

```
uint16_t RAIL_TxChannelHoppingConfig_t::bufferLength
```

This parameter must be set to the length of the buffer array, in 32 bit words.

This way, during configuration, the software can confirm it's writing within the bounds of the buffer. The configuration API will return an error or trigger [RAIL_ASSERT_CHANNEL_HOPPING_BUFFER_TOO_SHORT](#) if `bufferLength` is insufficient.

Definition at line 3003 of file `common/rail_types.h`

numberOfChannels

```
uint8_t RAIL_TxChannelHoppingConfig_t::numberOfChannels
```

The number of channels in the channel hopping sequence, which is the number of elements in the array that `entries` points to.

Definition at line 3008 of file `common/rail_types.h`

reserved

```
uint8_t RAIL_TxChannelHoppingConfig_t::reserved
```

Pad byte reserved for future use and currently ignored.

Definition at line 3012 of file `common/rail_types.h`

entries

```
RAIL_TxChannelHoppingConfigEntry_t* RAIL_TxChannelHoppingConfig_t::entries
```

A pointer to the first element of an array of [RAIL_TxChannelHoppingConfigEntry_t](#) that represents the channels used during channel hopping.

The length of this array must be `numberOfChannels`.

Definition at line 3019 of file `common/rail_types.h`

Thermal Protection

Thermal Protection

Modules

- [RAIL_ChipTempConfig_t](#)
- [RAIL_ChipTempMetrics_t](#)

Functions

- [RAIL_Status_t](#) [RAIL_ConfigThermalProtection](#)(RAIL_Handle_t genericRailHandle, const RAIL_ChipTempConfig_t *chipTempConfig)
Enable or disable the thermal protection if [RAIL_SUPPORTS_THERMAL_PROTECTION](#) is defined and update the temperature threshold and cool down hysteresis preventing or allowing transmissions.
- [RAIL_Status_t](#) [RAIL_GetThermalProtection](#)(RAIL_Handle_t genericRailHandle, RAIL_ChipTempConfig_t *chipTempConfig)
Get the current thermal configuration parameter and status.
- [RAIL_Status_t](#) [RAIL_GetTemperature](#)(RAIL_Handle_t railHandle, int16_t tempBuffer[((3U)+(1U))], bool reset)
Get the different temperature measurements in Kelvin done by sequencer or host.

Macros

- `#define` `RAIL_HFXO_TEMP_MEASURE_COUNT (1U)`
Number of temperature values provided for HFXO metrics.
- `#define` `RAIL_TEMP_MEASURE_COUNT undefined`
Total number of temperature values provided by [RAIL_GetTemperature\(\)](#).
- `#define` `RAIL_CHIP_TEMP_THRESHOLD_MAX (398U)`
Maximum junction temperature in Kelvin.
- `#define` `RAIL_CHIP_TEMP_COOLDOWN_DEFAULT (7U)`
Default number of Kelvin degrees below threshold needed to allow transmissions.
- `#define` `RAIL_CHIP_TEMP_MEASURE_COUNT (3U)`
Number of temperature values provided for the chip thermal protection.

Function Documentation

RAIL_ConfigThermalProtection

```
RAIL_Status_t RAIL_ConfigThermalProtection (RAIL_Handle_t genericRailHandle, const RAIL_ChipTempConfig_t *chipTempConfig)
```

Enable or disable the thermal protection if [RAIL_SUPPORTS_THERMAL_PROTECTION](#) is defined and update the temperature threshold and cool down hysteresis preventing or allowing transmissions.

Parameters

[in]	genericRailHandle	A radio-generic RAIL handle.
[in]	chipTempConfig	A pointer to a RAIL_ChipTempConfig_t that contains the configuration to be applied.

Returns

- Status code indicating the result of the function call. Returns `RAIL_STATUS_INVALID_PARAMETER` if enable field from `RAIL_ChipTempConfig_t` is set to false when an EFF is present on the board.
- When the temperature threshold minus a precise number of degrees specified by the cool down hysteresis parameter is exceeded, any future transmits are blocked until the temperature decreases below that limit. Besides, if the temperature threshold is exceeded, any active transmit is aborted.
- By default the threshold is set to `RAIL_CHIP_TEMP_THRESHOLD_MAX` and the cool down hysteresis is set to `RAIL_CHIP_TEMP_COOLDOWN_DEFAULT`.
- Note
- The thermal protection is automatically enabled when an EFF is present on the board. There is no use in calling this API in this case.

Definition at line 6499 of file common/rail.h

RAIL_GetThermalProtection

```
RAIL_Status_t RAIL_GetThermalProtection (RAIL_Handle_t genericRailHandle, RAIL_ChipTempConfig_t * chipTempConfig)
```

Get the current thermal configuration parameter and status.

Parameters

[in]	genericRailHandle	A radio-generic RAIL handle.
[out]	chipTempConfig	A non-NULL pointer to a <code>RAIL_ChipTempConfig_t</code> that will be updated with the current configuration.

Returns

- Status code indicating the result of the function call.

Definition at line 6510 of file common/rail.h

RAIL_GetTemperature

```
RAIL_Status_t RAIL_GetTemperature (RAIL_Handle_t railHandle, int16_t tempBuffer, bool reset)
```

Get the different temperature measurements in Kelvin done by sequencer or host.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	tempBuffer	The address of the array that will contain temperatures. This array must have at least <code>RAIL_TEMP_MEASURE_COUNT</code> entries.
[in]	reset	true to reset the temperature statistics, false otherwise.

Returns

- Status code indicating success of the function call.
- Temperatures, in Kelvin, are stored in tempBuffer such as:
- tempBuffer[0] is the chip temperature
 - tempBuffer[1] is the minimal chip temperature

tempBuffer[2] is the maximal chip temperature

Values that are not populated yet or incorrect are set to 0.

If [RAIL_SUPPORTS_HFXO_COMPENSATION](#) tempBuffer[3] is the HFXO temperature

Definition at line 6543 of file common/rail.h

RAIL_ChipTempConfig_t

Configuration parameters for thermal protection.

Public Attributes

bool	enable	Indicates whether the protection is enabled.
uint8_t	coolDownK	Mandatory temperature cool down when the threshold is exceeded, in degrees Kelvin.
uint16_t	thresholdK	Temperature above which transmit is blocked, in degrees Kelvin.

Public Attribute Documentation

enable

```
bool RAIL_ChipTempConfig_t::enable
```

Indicates whether the protection is enabled.

Definition at line 5796 of file `common/rail_types.h`

coolDownK

```
uint8_t RAIL_ChipTempConfig_t::coolDownK
```

Mandatory temperature cool down when the threshold is exceeded, in degrees Kelvin.

Definition at line 5798 of file `common/rail_types.h`

thresholdK

```
uint16_t RAIL_ChipTempConfig_t::thresholdK
```

Temperature above which transmit is blocked, in degrees Kelvin.

Definition at line 5800 of file `common/rail_types.h`

RAIL_ChipTempMetrics_t

Data used for thermal protection.

Public Attributes

uint16_t	tempK	Store chip temperature for metrics.
uint16_t	minTempK	Minimum temperature recorded.
uint16_t	maxTempK	Maximum temperature recorded.
bool	resetPending	Indicates if data should be reset.
uint8_t	reservedChipTemp	Reserved for future use.

Public Attribute Documentation

tempK

uint16_t RAIL_ChipTempMetrics_t::tempK

Store chip temperature for metrics.

Definition at line 5812 of file common/rail_types.h

minTempK

uint16_t RAIL_ChipTempMetrics_t::minTempK

Minimum temperature recorded.

Definition at line 5814 of file common/rail_types.h

maxTempK

uint16_t RAIL_ChipTempMetrics_t::maxTempK

Maximum temperature recorded.

Definition at line 5816 of file common/rail_types.h

resetPending

```
bool RAIL_ChipTempMetrics_t::resetPending
```

Indicates if data should be reset.

Definition at line 5818 of file common/rail_types.h

reservedChipTemp

```
uint8_t RAIL_ChipTempMetrics_t::reservedChipTemp
```

Reserved for future use.

Definition at line 5820 of file common/rail_types.h

Transmit

Transmit

APIs related to transmitting data packets.

Modules

[RAIL_TxPacketDetails_t](#)

[RAIL_ScheduleTxConfig_t](#)

[RAIL_CsmaConfig_t](#)

[RAIL_LbtConfig_t](#)

[RAIL_SyncWordConfig_t](#)

[RAIL_TxRepeatConfig_t](#)

[Packet Transmit](#)

[Power Amplifier \(PA\)](#)

Enumerations

```
enum RAIL\_StopMode\_t {  
    RAIL_STOP_MODE_ACTIVE_SHIFT = 0  
    RAIL_STOP_MODE_PENDING_SHIFT = 1  
}  
Stop radio operation options bit mask.  
  
enum RAIL\_TxOptions\_t {  
    RAIL_TX_OPTION_WAIT_FOR_ACK_SHIFT = 0  
    RAIL_TX_OPTION_REMOVE_CRC_SHIFT = 1  
    RAIL_TX_OPTION_SYNC_WORD_ID_SHIFT = 2  
    RAIL_TX_OPTION_ANTENNA0_SHIFT = 3  
    RAIL_TX_OPTION_ANTENNA1_SHIFT = 4  
    RAIL_TX_OPTION_ALT_PREAMBLE_LEN_SHIFT = 5  
    RAIL_TX_OPTION_CCA_PEAK_RSSI_SHIFT = 6  
    RAIL_TX_OPTION_CCA_ONLY_SHIFT = 7  
    RAIL_TX_OPTION_RESEND_SHIFT = 8  
    RAIL_TX_OPTION_CONCURRENT_PHY_ID_SHIFT = 9  
    RAIL_TX_OPTIONS_COUNT  
}  
Transmit options, in reality a bitmask.  
  
enum RAIL\_ScheduledTxDuringRx\_t {  
    RAIL_SCHEDULED_TX_DURING_RX_POSTPONE_TX = 0  
    RAIL_SCHEDULED_TX_DURING_RX_ABORT_TX = 1  
}  
Enumerates the possible outcomes of what will occur if a scheduled TX ends up firing during RX.  
  
enum RAIL\_TxRepeatOptions\_t {  
    RAIL_TX_REPEAT_OPTION_HOP_SHIFT = 0  
    RAIL_TX_REPEAT_OPTION_START_TO_START_SHIFT = 1
```



```
}
Transmit repeat
options, in reality a
bitmask.
```

Functions

RAIL_Status_t	RAIL_StopTx (RAIL_Handle_t railHandle, RAIL_StopMode_t mode) Stop an active or pending transmit.
RAIL_Status_t	RAIL_SetCcaThreshold (RAIL_Handle_t railHandle, int8_t ccaThresholdDbm) Set the CCA threshold in dBm.
RAIL_Status_t	RAIL_GetTxPacketDetails (RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails) Get detailed information about the last packet transmitted.
RAIL_Status_t	RAIL_GetTxPacketDetailsAlt (RAIL_Handle_t railHandle, bool isAck, RAIL_Time_t *pPacketTime) Get detailed information about the last packet transmitted.
RAIL_Status_t	RAIL_GetTxPacketDetailsAlt2 (RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails) Get detailed information about the last packet transmitted.
RAIL_Status_t	RAIL_GetTxTimePreambleStart (RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime) Adjust a RAIL TX completion timestamp to refer to the start of the preamble.
RAIL_Status_t	RAIL_GetTxTimePreambleStartAlt (RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails) Adjust a RAIL TX completion timestamp to refer to the start of the preamble.
RAIL_Status_t	RAIL_GetTxTimeSyncWordEnd (RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime) Adjust a RAIL TX timestamp to refer to the end of the sync word.
RAIL_Status_t	RAIL_GetTxTimeSyncWordEndAlt (RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails) Adjust a RAIL TX timestamp to refer to the end of the sync word.
RAIL_Status_t	RAIL_GetTxTimeFrameEnd (RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime) Adjust a RAIL TX timestamp to refer to the end of frame.
RAIL_Status_t	RAIL_GetTxTimeFrameEndAlt (RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails) Adjust a RAIL TX timestamp to refer to the end of frame.
RAIL_Status_t	RAIL_EnableTxHoldOff (RAIL_Handle_t railHandle, bool enable) Prevent the radio from starting a transmit.
bool	RAIL_IsTxHoldOffEnabled (RAIL_Handle_t railHandle) Check whether or not TX hold off is enabled.
RAIL_Status_t	RAIL_SetTxAltPreambleLength (RAIL_Handle_t railHandle, uint16_t length) Set an alternate transmitter preamble length.

Macros

#define	RAIL_STOP_MODES_NONE (0U) Do not stop any radio operations.
#define	RAIL_STOP_MODE_ACTIVE (1U << RAIL_STOP_MODE_ACTIVE_SHIFT) Stop active radio operations only.

```

#define RAIL_STOP_MODE_PENDING (1U << RAIL_STOP_MODE_PENDING_SHIFT)
    Stop pending radio operations.

#define RAIL_STOP_MODES_ALL (0xFFU)
    Stop all radio operations.

#define RAIL_TX_OPTIONS_NONE 0UL
    A value representing no options enabled.

#define RAIL_TX_OPTIONS_DEFAULT RAIL_TX_OPTIONS_NONE
    All options disabled by default.

#define RAIL_TX_OPTION_WAIT_FOR_ACK (1UL << RAIL_TX_OPTION_WAIT_FOR_ACK_SHIFT)
    An option when Auto-Ack has been configured, enabled, and not TX paused, to configure whether or not the
    transmitting node will listen for an Ack response.

#define RAIL_TX_OPTION_REMOVE_CRC (1UL << RAIL_TX_OPTION_REMOVE_CRC_SHIFT)
    An option to remove CRC bytes from TX packets.

#define RAIL_TX_OPTION_SYNC_WORD_ID (1UL << RAIL_TX_OPTION_SYNC_WORD_ID_SHIFT)
    An option to select which sync word is used during the transmission.

#define RAIL_TX_OPTION_ANTENNA0 (1UL << RAIL_TX_OPTION_ANTENNA0_SHIFT)
    An option to select antenna 0 for transmission.

#define RAIL_TX_OPTION_ANTENNA1 (1UL << RAIL_TX_OPTION_ANTENNA1_SHIFT)
    An option to select antenna 1 for transmission.

#define RAIL_TX_OPTION_ALT_PREAMBLE_LEN (1UL << RAIL_TX_OPTION_ALT_PREAMBLE_LEN_SHIFT)
    An option to use the alternate preamble length established by RAIL\_SetTxAltPreambleLength\(\) for the transmission.

#define RAIL_TX_OPTION_CCA_PEAK_RSSI (1UL << RAIL_TX_OPTION_CCA_PEAK_RSSI_SHIFT)
    An option to use peak rather than average RSSI energy detected during CSMA's RAIL\_CsmaConfig\_t::ccaDuration or
    LBT's RAIL\_LbtConfig\_t::lbtDuration to determine whether the channel is clear or busy.

#define RAIL_TX_OPTION_CCA_ONLY (1UL << RAIL_TX_OPTION_CCA_ONLY_SHIFT)
    An option to only perform the CCA (CSMA/LBT) operation but not automatically transmit if the channel is clear.

#define RAIL_TX_OPTION_RESEND (1UL << RAIL_TX_OPTION_RESEND_SHIFT)
    An option to resend packet at the beginning of the Transmit FIFO.

#define RAIL_TX_OPTION_CONCURRENT_PHY_ID (1UL << RAIL_TX_OPTION_CONCURRENT_PHY_ID_SHIFT)
    An option to specify which PHY is used to transmit in the case of concurrent mode.

#define RAIL_TX_OPTIONS_ALL 0xFFFFFFFFUL
    A value representing all possible options.

#define RAIL_MAX_LBT_TRIES (15U)
    The maximum number of LBT/CSMA retries supported.

#define RAIL_MAX_CSMA_EXPONENT (8U)
    The maximum power-of-2 exponent for CSMA backoffs.

#define RAIL_CSMA_CONFIG_802_15_4_2003_2p4_GHz_OQPSK_CSMA undefined
    RAIL\_CsmaConfig\_t initializer configuring CSMA per IEEE 802.15.4-2003 on 2.4 GHz OQPSK, commonly used by Zigbee.

#define RAIL_CSMA_CONFIG_SINGLE_CCA undefined
    RAIL\_CsmaConfig\_t initializer configuring a single CCA prior to TX.

#define RAIL_LBT_CONFIG_ETSI_EN_300_220_1_V2_4_1 undefined
    RAIL\_LbtConfig\_t initializer configuring LBT per ETSI 300 220-1 V2.4.1 for a typical Sub-GHz band.

```

```
#define RAIL_LBT_CONFIG_ETSI_EN_300_220_1_V3_1_0 undefined
RAIL_LbtConfig_t initializer configuring LBT per ETSI 300 220-1 V3.1.0 for a typical Sub-GHz band.

#define RAIL_TX_REPEAT_OPTIONS_NONE 0U
A value representing no repeat options enabled.

#define RAIL_TX_REPEAT_OPTIONS_DEFAULT RAIL_TX_REPEAT_OPTIONS_NONE
All repeat options disabled by default.

#define RAIL_TX_REPEAT_OPTION_HOP (1U << RAIL_TX_REPEAT_OPTION_HOP_SHIFT)
An option to configure whether or not to channel-hop before each repeated transmit.

#define RAIL_TX_REPEAT_OPTION_START_TO_START (1 << RAIL_TX_REPEAT_OPTION_START_TO_START_SHIFT)
An option to configure the delay between transmissions to be from start to start instead of end to start.

#define RAIL_TX_REPEAT_INFINITE_ITERATIONS (0xFFFFU)
RAIL_TxRepeatConfig_t::iterations initializer configuring infinite repeated transmissions.
```

Enumeration Documentation

RAIL_StopMode_t

RAIL_StopMode_t

Stop radio operation options bit mask.

Enumerator	
RAIL_STOP_MODE_ACTIVE_SHIFT	Shift position of RAIL_STOP_MODE_ACTIVE bit.
RAIL_STOP_MODE_PENDING_SHIFT	Shift position of RAIL_STOP_MODE_PENDING bit.

Definition at line 3041 of file `common/rail_types.h`

RAIL_TxOptions_t

RAIL_TxOptions_t

Transmit options, in reality a bitmask.

Enumerator	
RAIL_TX_OPTION_WAIT_FOR_ACK_SHIFT	Shift position of RAIL_TX_OPTION_WAIT_FOR_ACK bit.
RAIL_TX_OPTION_REMOVE_CRC_SHIFT	Shift position of RAIL_TX_OPTION_REMOVE_CRC bit.
RAIL_TX_OPTION_SYNC_WORD_ID_SHIFT	Shift position of RAIL_TX_OPTION_SYNC_WORD_ID bit.
RAIL_TX_OPTION_ANTENNA0_SHIFT	Shift position of RAIL_TX_OPTION_ANTENNA0 bit.
RAIL_TX_OPTION_ANTENNA1_SHIFT	Shift position of RAIL_TX_OPTION_ANTENNA1 bit.
RAIL_TX_OPTION_ALT_PREAMBLE_LEN_SHIFT	Shift position of RAIL_TX_OPTION_ALT_PREAMBLE_LEN bit.
RAIL_TX_OPTION_CCA_PEAK_RSSI_SHIFT	Shift position of RAIL_TX_OPTION_CCA_PEAK_RSSI bit.
RAIL_TX_OPTION_CCA_ONLY_SHIFT	Shift position of RAIL_TX_OPTION_CCA_ONLY bit.
RAIL_TX_OPTION_RESEND_SHIFT	Shift position of RAIL_TX_OPTION_RESEND bit.
RAIL_TX_OPTION_CONCURRENT_PHY_ID_SHIFT	Shift position of RAIL_TX_OPTION_CONCURRENT_PHY_ID bit.
RAIL_TX_OPTIONS_COUNT	A count of the choices in this enumeration.

Definition at line 3061 of file `common/rail_types.h`

RAIL_ScheduledTxDuringRx_t

RAIL_ScheduledTxDuringRx_t

Enumerates the possible outcomes of what will occur if a scheduled TX ends up firing during RX.

Because RX and TX can't happen at the same time, it is up to the user how the TX should be handled. This enumeration is passed into [RAIL_StartScheduledTx\(\)](#) as part of [RAIL_ScheduleTxConfig_t](#).

Enumerator	
RAIL_SCHEDULED_TX_DURING_RX_POSTPONE_TX	The scheduled TX will be postponed until RX completes and then sent.
RAIL_SCHEDULED_TX_DURING_RX_ABORT_TX	The scheduled TX will be aborted and a RAIL_EVENT_TX_BLOCKED event will fire.

Definition at line 3247 of file `common/rail_types.h`

RAIL_TxRepeatOptions_t

RAIL_TxRepeatOptions_t

Transmit repeat options, in reality a bitmask.

Enumerator	
RAIL_TX_REPEAT_OPTION_HOP_SHIFT	Shift position of RAIL_TX_REPEAT_OPTION_HOP bit.
RAIL_TX_REPEAT_OPTION_START_TO_START_SHIFT	Shift position of the RAIL_TX_REPEAT_OPTION_START_TO_START bit.

Definition at line 3620 of file `common/rail_types.h`

Function Documentation

RAIL_StopTx

RAIL_Status_t RAIL_StopTx ([RAIL_Handle_t](#) railHandle, [RAIL_StopMode_t](#) mode)

Stop an active or pending transmit.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mode	Specifies the type(s) of transmit to stop.

Returns

- [RAIL_STATUS_NO_ERROR](#) if the transmit was successfully stopped or [RAIL_STATUS_INVALID_STATE](#) if there is no transmit operation to stop.

Note

- When mode includes [RAIL_STOP_MODE_ACTIVE](#), this can also stop an active Auto-Ack transmit. When an active transmit is stopped, [RAIL_EVENT_TX_ABORTED](#) or [RAIL_EVENT_TXACK_ABORTED](#) should occur. When mode includes [RAIL_STOP_MODE_PENDING](#) this can also stop a [RAIL_TX_OPTION_CCA_ONLY](#) transmit operation. When a pending transmit is stopped, [RAIL_EVENT_TX_BLOCKED](#) should occur.

Definition at line 3444 of file `common/rail.h`

RAIL_SetCcaThreshold

```
RAIL_Status_t RAIL_SetCcaThreshold (RAIL_Handle_t railHandle, int8_t ccaThresholdDbm)
```

Set the CCA threshold in dBm.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	ccaThresholdDbm	The CCA threshold in dBm.

Returns

- Status code indicating success of the function call.
- Unlike [RAIL_StartCcaCsmaTx\(\)](#) or [RAIL_StartCcaLbtTx\(\)](#), which can cause a transmit, this function only modifies the CCA threshold. A possible use case for this function involves setting the CCA threshold to invalid RSSI of -128 which blocks transmission by preventing clear channel assessments from succeeding.

Definition at line 3459 of file `common/rail.h`

RAIL_GetTxPacketDetails

```
RAIL_Status_t RAIL_GetTxPacketDetails (RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t * pPacketDetails)
```

Get detailed information about the last packet transmitted.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	An application-provided pointer to store RAIL_TxPacketDetails_t corresponding to the transmit event. The isAck and timeSent fields totalPacketBytes and timePosition must be initialized prior to each call: • isAck true to obtain details about the most recent Ack transmit, false to obtain details about the most recent app-initiated transmit. • totalPacketBytes with the total number of bytes of the transmitted packet for RAIL to use when calculating the specified timestamp. This should account for all bytes sent over the air after the Preamble and Sync word(s), including CRC bytes. • timePosition with a RAIL_PacketTimePosition_t value specifying the packet position to put in the timeSent field on return. This field will also be updated with the actual position corresponding to the timeSent value filled in.

Returns

- [RAIL_STATUS_NO_ERROR](#) if pPacketDetails was filled in, or an appropriate error code otherwise.

Note

- Consider using [RAIL_GetTxPacketDetailsAlt2](#) for smaller code size.
- This function can only be called from callback context for either [RAIL_EVENT_TX_PACKET_SENT](#) or [RAIL_EVENT_TXACK_PACKET_SENT](#) events.

Definition at line 3489 of file `common/rail.h`

RAIL_GetTxPacketDetailsAlt

RAIL_Status_t RAIL_GetTxPacketDetailsAlt (RAIL_Handle_t railHandle, bool isAck, RAIL_Time_t * pPacketTime)

Get detailed information about the last packet transmitted.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	isAck	true to obtain details about the most recent Ack transmit. false to obtain details about the most recent app-initiated transmit.
[out]	pPacketTime	An application-provided non-NULL pointer to store a RAIL_Time_t corresponding to the transmit event. This will be populated with a timestamp corresponding to an arbitrary location in the packet. Call RAIL_GetTxTimePreambleStart , RAIL_GetTxTimeSyncWordEnd , or RAIL_GetTxTimeFrameEnd to adjust the timestamp for different locations in the packet.

Returns

- [RAIL_STATUS_NO_ERROR](#) if pPacketTime was filled in, or an appropriate error code otherwise.

Note

- Consider using [RAIL_GetTxPacketDetailsAlt2](#) to pass in a [RAIL_PacketTimeStamp_t](#) structure instead of a [RAIL_Time_t](#) structure, particularly when [RAIL_PacketTimePosition_t](#) information is needed or useful.

This function can only be called from callback context for either [RAIL_EVENT_TX_PACKET_SENT](#) or [RAIL_EVENT_TXACK_PACKET_SENT](#) events.

Definition at line 3516 of file common/rail.h

RAIL_GetTxPacketDetailsAlt2

RAIL_Status_t RAIL_GetTxPacketDetailsAlt2 (RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t * pPacketDetails)

Get detailed information about the last packet transmitted.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	An application-provided pointer to store RAIL_TxPacketDetails_t corresponding to the transmit event. The isAck field must be initialized prior to each call: • isAck true to obtain details about the most recent Ack transmit, false to obtain details about the most recent app-initiated transmit. The timeSent field packetTime will be populated with a timestamp corresponding to a default location in the packet. The timeSent field timePosition will be populated with a RAIL_PacketTimePosition_t value specifying that default packet location. Call RAIL_GetTxTimePreambleStartAlt(), RAIL_GetTxTimeSyncWordEndAlt(), or RAIL_GetTxTimeFrameEndAlt() to adjust the timestamp for different locations in the packet.

Returns

- [RAIL_STATUS_NO_ERROR](#) if pPacketDetails was filled in, or an appropriate error code otherwise.

This function can only be called from callback context for either [RAIL_EVENT_TX_PACKET_SENT](#) or [RAIL_EVENT_TXACK_PACKET_SENT](#) events.

Definition at line 3543 of file common/rail.h

RAIL_GetTxTimePreambleStart

```
RAIL_Status_t RAIL_GetTxTimePreambleStart (RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t * pPacketTime)
```

Adjust a RAIL TX completion timestamp to refer to the start of the preamble.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	totalPacketBytes	The total number of bytes of the transmitted packet for RAIL to use when adjusting the provided timestamp. This should account for all bytes transmitted over the air after the Preamble and Sync word(s), including CRC bytes. Pass RAIL_TX_STARTED_BYTES to retrieve the start-of-normal-TX timestamp (see below).
[inout]	pPacketTime	This points to the RAIL_Time_t returned from a previous call to RAIL_GetTxPacketDetailsAlt() for this same packet. That time will be updated with the time that the preamble for this packet started on air. Must be non-NULL.

Also used to retrieve the [RAIL_EVENT_TX_STARTED](#) timestamp.

Returns

- [RAIL_STATUS_NO_ERROR](#) if pPacketTime was successfully determined or an appropriate error code otherwise.

When used for timestamp adjustment, call this function in the same transmit-complete event-handling context as [RAIL_GetTxPacketDetailsAlt\(\)](#) is called.

This function may be called when handling the [RAIL_EVENT_TX_STARTED](#) event to retrieve that event's start-of-normal-TX timestamp. (Ack transmits currently have no equivalent event or associated timestamp.) In this case, totalPacketBytes must be [RAIL_TX_STARTED_BYTES](#), and pPacketTime is an output-only parameter filled in with that time (so no need to initialize it beforehand by calling [RAIL_GetTxPacketDetailsAlt\(\)](#)).

Definition at line 3578 of file `common/rail.h`

RAIL_GetTxTimePreambleStartAlt

```
RAIL_Status_t RAIL_GetTxTimePreambleStartAlt (RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t * pPacketDetails)
```

Adjust a RAIL TX completion timestamp to refer to the start of the preamble.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	A non-NULL pointer to the details that were returned from a previous call to RAIL_GetTxPacketDetailsAlt2() for this same packet. The application must update the timeSent field totalPacketBytes to be the total number of bytes of the sent packet for RAIL to use when calculating the specified timestamp. This should account for all bytes transmitted over the air after the Preamble and Sync word(s), including CRC bytes. Pass RAIL_TX_STARTED_BYTES to retrieve the start-of-normal-TX timestamp (see below). After this function, the timeSent field packetTime will be updated with the time that the preamble for this packet started on air.

Also used to retrieve the [RAIL_EVENT_TX_STARTED](#) timestamp.

Returns

- [RAIL_STATUS_NO_ERROR](#) if the packet time was successfully calculated, or an appropriate error code otherwise.

When used for timestamp adjustment, call this function in the same transmit-complete event-handling context as [RAIL_GetTxPacketDetailsAlt2\(\)](#) is called.

This function may be called when handling the [RAIL_EVENT_TX_STARTED](#) event to retrieve that event's start-of-normal-TX timestamp. (Ack transmits currently have no equivalent event or associated timestamp.) In this case, the timeSent field totalPacketBytes must be [RAIL_TX_STARTED_BYTES](#), and the timeSent field packetTime is an output-only parameter filled in with that time (so no need to initialize it beforehand by calling [RAIL_GetTxPacketDetailsAlt2\(\)](#)).

Definition at line 3613 of file common/rail.h

RAIL_GetTxTimeSyncWordEnd

RAIL_Status_t RAIL_GetTxTimeSyncWordEnd ([RAIL_Handle_t](#) railHandle, uint16_t totalPacketBytes, [RAIL_Time_t](#) * pPacketTime)

Adjust a RAIL TX timestamp to refer to the end of the sync word.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	totalPacketBytes	The total number of bytes of the transmitted packet for RAIL to use when calculating the specified timestamp. This should account for all bytes transmitted over the air after the Preamble and Sync word(s), including CRC bytes.
[inout]	pPacketTime	The time that was returned in a RAIL_Time_t from a previous call to RAIL_GetTxPacketDetailsAlt() for this same packet. After this function, the time at that location will be updated with the time that the sync word for this packet finished on air. Must be non-NULL.

Returns

- [RAIL_STATUS_NO_ERROR](#) if pPacketTime was successfully calculated, or an appropriate error code otherwise.

Call the timestamp adjustment function in the same transmit-complete event-handling context as [RAIL_GetTxPacketDetailsAlt\(\)](#) is called.

Definition at line 3636 of file common/rail.h

RAIL_GetTxTimeSyncWordEndAlt

RAIL_Status_t RAIL_GetTxTimeSyncWordEndAlt ([RAIL_Handle_t](#) railHandle, [RAIL_TxPacketDetails_t](#) * pPacketDetails)

Adjust a RAIL TX timestamp to refer to the end of the sync word.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	A non-NULL pointer to the details that were returned from a previous call to RAIL_GetTxPacketDetailsAlt2() for this same packet. The application must update the timeSent field totalPacketBytes to be the total number of bytes of the sent packet for RAIL to use when calculating the specified timestamp. This should account for all bytes transmitted over the air after the Preamble and Sync word(s), including CRC bytes. After this function, the timeSent field packetTime will be updated with the time that the sync word for this packet finished on air.

Returns

- [RAIL_STATUS_NO_ERROR](#) if the packet time was successfully calculated, or an appropriate error code otherwise.

Call the timestamp adjustment function in the same transmit-complete event-handling context as [RAIL_GetTxPacketDetailsAlt2\(\)](#) is called.

Definition at line 3660 of file `common/rail.h`

RAIL_GetTxTimeFrameEnd

```
RAIL_Status_t RAIL_GetTxTimeFrameEnd (RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t * pPacketTime)
```

Adjust a RAIL TX timestamp to refer to the end of frame.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	totalPacketBytes	The total number of bytes of the transmitted packet for RAIL to use when calculating the specified timestamp. This should account for all bytes transmitted over the air after the Preamble and Sync word(s), including CRC bytes.
[inout]	pPacketTime	The time that was returned in a RAIL_Time_t from a previous call to RAIL_GetTxPacketDetailsAlt() for this same packet. After this function, the time at that location will be updated with the time that this packet finished on air. Must be non-NULL.

Returns

- [RAIL_STATUS_NO_ERROR](#) if pPacketTime was successfully calculated, or an appropriate error code otherwise.

Call the timestamp adjustment function in the same transmit-complete event-handling context as [RAIL_GetTxPacketDetailsAlt\(\)](#) is called.

Definition at line 3683 of file `common/rail.h`

RAIL_GetTxTimeFrameEndAlt

```
RAIL_Status_t RAIL_GetTxTimeFrameEndAlt (RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t * pPacketDetails)
```

Adjust a RAIL TX timestamp to refer to the end of frame.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	A non-NULL pointer to the details that were returned from a previous call to RAIL_GetTxPacketDetailsAlt2() for this same packet. The application must update the timeSent field totalPacketBytes to be the total number of bytes of the sent packet for RAIL to use when calculating the specified timestamp. This should account for all bytes transmitted over the air after the Preamble and Sync word(s), including CRC bytes. After this function, the timeSent field packetTime will be updated with the time that this packet finished on air.

Returns

- [RAIL_STATUS_NO_ERROR](#) if the packet time was successfully calculated, or an appropriate error code otherwise.

Call the timestamp adjustment function in the same transmit-complete event-handling context as [RAIL_GetTxPacketDetailsAlt2\(\)](#) is called.

Definition at line 3706 of file `common/rail.h`

RAIL_EnableTxHoldOff

RAIL_Status_t RAIL_EnableTxHoldOff ([RAIL_Handle_t](#) railHandle, bool enable)

Prevent the radio from starting a transmit.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	Enable/Disable TX hold off.

Returns

- Status code indicating success of the function call.

Enable TX hold off to prevent the radio from starting any transmits. Disable TX hold off to allow the radio to transmit again. Attempting to transmit with the TX hold off enabled will result in [RAIL_EVENT_TX_BLOCKED](#) and/or [RAIL_EVENT_TXACK_BLOCKED](#) events.

Note

- This function does not affect a transmit that has already started. To stop an already-started transmission, use [RAIL_Idle\(\)](#) with [RAIL_IDLE_ABORT](#).

Definition at line 3726 of file `common/rail.h`

RAIL_IsTxHoldOffEnabled

bool RAIL_IsTxHoldOffEnabled ([RAIL_Handle_t](#) railHandle)

Check whether or not TX hold off is enabled.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if TX hold off is enabled, false otherwise.

TX hold off can be enabled/disabled using [RAIL_EnableTxHoldOff\(\)](#). Attempting to transmit with the TX hold off enabled will block the transmission and result in [RAIL_EVENT_TX_BLOCKED](#) and/or [RAIL_EVENT_TXACK_BLOCKED](#) events.

Definition at line 3739 of file `common/rail.h`

RAIL_SetTxAltPreambleLength

RAIL_Status_t RAIL_SetTxAltPreambleLength ([RAIL_Handle_t](#) railHandle, uint16_t length)

Set an alternate transmitter preamble length.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	length	The desired preamble length, in bits.

Returns

- Status code indicating success of the function call.

To cause a transmission to use this alternate preamble length, specify [RAIL_TX_OPTION_ALT_PREAMBLE_LEN](#) in the txOptions parameter passed to the respective RAIL transmit API.

Note

- Attempting to set a preamble length of 0xFFFF bits will result in [RAIL_STATUS_INVALID_PARAMETER](#).

Definition at line 3755 of file common/rail.h

RAIL_TxPacketDetails_t

Detailed information requested about the packet that was just, or is currently being, transmitted.

Public Attributes

RAIL_PacketTimeStamp_t	timeSent The timestamp of the transmitted packet in the RAIL timebase, filled in by RAIL_GetTxPacketDetails() .
bool	isAck Indicate whether the transmitted packet was an automatic Ack.

Public Attribute Documentation

timeSent

RAIL_PacketTimeStamp_t RAIL_TxPacketDetails_t::timeSent

The timestamp of the transmitted packet in the RAIL timebase, filled in by [RAIL_GetTxPacketDetails\(\)](#).

Definition at line 3228 of file `common/rail_types.h`

isAck

bool RAIL_TxPacketDetails_t::isAck

Indicate whether the transmitted packet was an automatic Ack.

In a generic sense, an automatic Ack is defined as a packet sent in response to a received Ack-requesting frame when Auto-Ack is enabled. In a protocol specific sense this definition may be more or less restrictive to match the specification and you should refer to that protocol's documentation.

Definition at line 3236 of file `common/rail_types.h`

RAIL_ScheduleTxConfig_t

A configuration structure for a scheduled transmit.

Public Attributes

RAIL_Time_t	when	The time when to transmit this packet.
RAIL_TimeMode_t	mode	The type of delay.
RAIL_ScheduledTxDuringRx_t	txDuringRx	Indicate which action to take with a scheduled TX if it occurs during RX.

Public Attribute Documentation

when

RAIL_Time_t RAIL_ScheduleTxConfig_t::when

The time when to transmit this packet.

The exact interpretation of this value depends on the mode specified below.

Definition at line 3274 of file common/rail_types.h

mode

RAIL_TimeMode_t RAIL_ScheduleTxConfig_t::mode

The type of delay.

See the [RAIL_TimeMode_t](#) documentation for more information. Be sure to use [RAIL_TIME_ABSOLUTE](#) delays for time-critical protocols.

Definition at line 3280 of file common/rail_types.h

txDuringRx

RAIL_ScheduledTxDuringRx_t RAIL_ScheduleTxConfig_t::txDuringRx

Indicate which action to take with a scheduled TX if it occurs during RX.

See [RAIL_ScheduledTxDuringRx_t](#) structure for more information on potential options.

Definition at line 3286 of file common/rail_types.h

RAIL_CsmaConfig_t

A configuration structure for the CSMA transmit algorithm.

One of RAIL's schemes for polite spectrum access is an implementation of a Carrier-Sense Multiple Access (CSMA) algorithm based on IEEE 802.15.4 (unslotted).

In pseudo-code it works like this, showing relevant event notifications:

```

// Return true to transmit packet, false to not transmit packet.
bool performCsma(const RAIL_CsmaConfig_t *csmaConfig)
{
    bool isFixedBackoff = ((csmaConfig->csmaMinBoExp == 0)
        && (csmaConfig->csmaMaxBoExp == 0));
    int backoffExp = csmaConfig->csmaMinBoExp; // Initial backoff exponent
    int backoffMultiplier = 1; // Assume fixed backoff
    int try;

    // Special-case tries == 0 to transmit immediately without backoff+CCA
    if (csmaConfig->csmaTries == 0) {
        return true;
    }

    // Start overall timeout if specified:
    if (csmaConfig->csmaTimeout > 0) {
        StartAbortTimer(csmaConfig->csmaTimeout, RAIL_EVENT_TX_CHANNEL_BUSY);
        // If timeout occurs, abort and signal the indicated event.
    }

    for (try = 0; try < csmaConfig->csmaTries; try++) {
        if (try > 0) {
            signalEvent(RAIL_EVENT_TX_CCA_RETRY);
        }
        // Determine the backoff multiplier for this try:
        if (isFixedBackoff) {
            // backoffMultiplier already set to 1 for fixed backoff
        } else {
            // Start with the backoff exponent for this try:
            if (try > 0) {
                backoffExp++;
                if (backoffExp > csmaConfig->csmaMaxBoExp) {
                    backoffExp = csmaConfig->csmaMaxBoExp;
                }
            }
            // Pick random multiplier between 0 and 2*backoffExp - 1 inclusive:
            backoffMultiplier = pickRandomInteger(0, (1 << backoffExp) - 1);
        }
        // Perform the backoff:
        delayMicroseconds(backoffMultiplier * csmaConfig->ccaBackoff);
        // Perform the Clear-Channel Assessment (CCA):
        // Channel is considered busy if radio is actively receiving or
        // transmitting, or the average energy detected across duration
        // is above the threshold.
        signalEvent(RAIL_EVENT_TX_START_CCA);
        if (performCca(csmaConfig->ccaDuration, csmaConfig->ccaThreshold)) {
            // CCA (and CSMA) success: Transmit after RX-to-TX turnaround
            StopAbortTimer();
            signalEvent(RAIL_EVENT_TX_CHANNEL_CLEAR);
            return true;
        } else {
            // CCA failed: loop to try again, or exit if out of tries
        }
    }
    // Overall CSMA failure: Don't transmit
    StopAbortTimer();
    signalEvent(RAIL_EVENT_TX_CHANNEL_BUSY);
    return false;
}

```

Public Attributes

uint8_t	csmaMinBoExp	The minimum (starting) exponent for CSMA random backoff ($2^{\text{exp}} - 1$).
uint8_t	csmaMaxBoExp	The maximum exponent for CSMA random backoff ($2^{\text{exp}} - 1$).
uint8_t	csmaTries	The number of backoff-then-CCA iterations that can fail before reporting RAIL_EVENT_TX_CHANNEL_BUSY .
int8_t	ccaThreshold	The CCA RSSI threshold, in dBm, above which the channel is considered 'busy'.
uint16_t	ccaBackoff	The backoff unit period in RAIL's microsecond time base.
uint16_t	ccaDuration	The minimum desired CCA check duration in microseconds.
RAIL_Time_t	csmaTimeout	An overall timeout, in RAIL's microsecond time base, for the operation.

Public Attribute Documentation

csmaMinBoExp

```
uint8_t RAIL_CsmaConfig_t::csmaMinBoExp
```

The minimum (starting) exponent for CSMA random backoff ($2^{\text{exp}} - 1$).

It can range from 0 to [RAIL_MAX_CSMA_EXPONENT](#).

Warnings

- On EFR32, due to a hardware limitation, this can only be 0 if [csmaMaxBoExp](#) is also 0 specifying a non-random fixed backoff. [RAIL_STATUS_INVALID_PARAMETER](#) will result otherwise. If you really want CSMA's first iteration to have no backoff prior to CCA, with subsequent iterations having random backoff as the exponent is increased, you must do a fixed backoff of 0 operation first ([csmaMinBoExp](#) = 0, [csmaMaxBoExp](#) = 0, [ccaBackoff](#) = 0, [csmaTries](#) = 1), and if that fails ([RAIL_EVENT_TX_CHANNEL_BUSY](#)), follow up with a random backoff operation starting at [csmaMinBoExp](#) = 1 for the remaining iterations.

Definition at line 3387 of file `common/rail_types.h`

csmaMaxBoExp

```
uint8_t RAIL_CsmaConfig_t::csmaMaxBoExp
```

The maximum exponent for CSMA random backoff ($2^{\text{exp}} - 1$).

It can range from 0 to [RAIL_MAX_CSMA_EXPONENT](#) and must be greater than or equal to [csmaMinBoExp](#). If both exponents are 0, a non-random fixed backoff of [ccaBackoff](#) duration results.

Definition at line 3395 of file `common/rail_types.h`

csmaTries


```
uint8_t RAIL_CsmaConfig_t::csmaTries
```

The number of backoff-then-CCA iterations that can fail before reporting [RAIL_EVENT_TX_CHANNEL_BUSY](#).

Typically ranges from 1 to [RAIL_MAX_LBT_TRIES](#); higher values are disallowed. A value 0 always transmits immediately without performing CSMA, similar to calling [RAIL_StartTx\(\)](#).

Definition at line 3403 of file `common/rail_types.h`

ccaThreshold

```
int8_t RAIL_CsmaConfig_t::ccaThreshold
```

The CCA RSSI threshold, in dBm, above which the channel is considered 'busy'.

Definition at line 3408 of file `common/rail_types.h`

ccaBackoff

```
uint16_t RAIL_CsmaConfig_t::ccaBackoff
```

The backoff unit period in RAIL's microsecond time base.

It is multiplied by the random backoff exponential controlled by [csmaMinBoExp](#) and [csmaMaxBoExp](#) to determine the overall backoff period. For random backoffs, any value above 32768 microseconds for the 'EFR Series 2' and 8192 microseconds for the 'Series 3' will be truncated for a single backoff period. Up to 255 backoff periods are supported. For fixed backoffs it can go up to 65535 microseconds.

Definition at line 3418 of file `common/rail_types.h`

ccaDuration

```
uint16_t RAIL_CsmaConfig_t::ccaDuration
```

The minimum desired CCA check duration in microseconds.

Note

- Depending on the radio configuration, due to hardware constraints, the actual duration may be longer. Also, if the requested duration is too large for the radio to accommodate, [RAIL_StartCcaCsmaTx\(\)](#) will fail returning [RAIL_STATUS_INVALID_PARAMETER](#).

Definition at line 3427 of file `common/rail_types.h`

csmaTimeout

```
RAIL_Time_t RAIL_CsmaConfig_t::csmaTimeout
```

An overall timeout, in RAIL's microsecond time base, for the operation.

If the transmission doesn't start before this timeout expires, the transmission will fail with [RAIL_EVENT_TX_CHANNEL_BUSY](#).
A value 0 means no timeout is imposed.

Definition at line 3434 of file `common/rail_types.h`

RAIL_LbtConfig_t

A configuration structure for the LBT transmit algorithm.

One of RAIL's schemes for polite spectrum access is an implementation of a Listen-Before-Talk (LBT) algorithm, loosely based on ETSI 300 220-1.

Currently, however, it is constrained by the EFR32's CSMA-oriented hardware so is turned into an equivalent [RAIL_CsmaConfig_t](#) configuration and passed to the CSMA engine:

```
if (lbtMaxBoRand == lbtMinBoRand) {
    // Fixed backoff
    csmaMinBoExp = csmaMaxBoExp = 0;
    if (lbtMinBoRand == 0) {
        ccaBackoff = lbtBackoff;
    } else {
        ccaBackoff = lbtMinBoRand * lbtBackoff;
    }
    ccaDuration = lbtDuration;
} else {
    // Random backoff: map to random range 0 .. (lbtMaxBoRand - lbtMinBoRand)
    csmaMinBoExp = csmaMaxBoExp = ceiling(log2(lbtMaxBoRand - lbtMinBoRand));
    ccaBackoff = round((lbtBackoff * (lbtMaxBoRand - lbtMinBoRand))
        / (1 << csmaMinBoExp));
    ccaDuration = lbtDuration + (lbtMinBoRand * lbtBackoff);
}
csmaTries = lbtTries;
ccaThreshold = lbtThreshold;
csmaTimeout = lbtTimeout;
```

Public Attributes

uint8_t	lbtMinBoRand	The minimum backoff random multiplier.
uint8_t	lbtMaxBoRand	The maximum backoff random multiplier.
uint8_t	lbtTries	The number of LBT iterations that can fail before reporting RAIL_EVENT_TX_CHANNEL_BUSY .
int8_t	lbtThreshold	The LBT RSSI threshold, in dBm, above which the channel is considered 'busy'.
uint16_t	lbtBackoff	The backoff unit period, in RAIL's microsecond time base.
uint16_t	lbtDuration	The minimum desired LBT check duration in microseconds.
RAIL_Time_t	lbtTimeout	An overall timeout, in RAIL's microsecond time base, for the operation.

Public Attribute Documentation

lbtMinBoRand

```
uint8_t RAIL_LbtConfig_t::lbtMinBoRand
```

The minimum backoff random multiplier.

Definition at line 3506 of file `common/rail_types.h`

lbtMaxBoRand

```
uint8_t RAIL_LbtConfig_t::lbtMaxBoRand
```

The maximum backoff random multiplier.

It must be greater than or equal to [lbtMinBoRand](#).

If both backoff multipliers are identical, a non-random fixed backoff of [lbtBackoff](#) times the multiplier (minimum 1) duration results.

Definition at line 3513 of file `common/rail_types.h`

lbtTries

```
uint8_t RAIL_LbtConfig_t::lbtTries
```

The number of LBT iterations that can fail before reporting [RAIL_EVENT_TX_CHANNEL_BUSY](#).

Typically ranges from 1 to [RAIL_MAX_LBT_TRIES](#); higher values are disallowed. A value 0 always transmits immediately without performing LBT, similar to calling [RAIL_StartTx\(\)](#).

Definition at line 3521 of file `common/rail_types.h`

lbtThreshold

```
int8_t RAIL_LbtConfig_t::lbtThreshold
```

The LBT RSSI threshold, in dBm, above which the channel is considered 'busy'.

Definition at line 3526 of file `common/rail_types.h`

lbtBackoff

```
uint16_t RAIL_LbtConfig_t::lbtBackoff
```

The backoff unit period, in RAIL's microsecond time base.

It is multiplied by the random backoff multiplier controlled by [lbtMinBoRand](#) and [lbtMaxBoRand](#) to determine the overall backoff period. For random backoffs, any value above 32768 microseconds for the 'EFR Series 2' and 8192 microseconds for the 'Series 3' will be truncated for a single backoff period. Up to 255 backoff periods are supported. For fixed backoffs, it can go up to 65535 microseconds.

Definition at line 3536 of file `common/rail_types.h`

lbtDuration

```
uint16_t RAIL_LbtConfig_t::lbtDuration
```

The minimum desired LBT check duration in microseconds.

Note

- Depending on the radio configuration, due to hardware constraints, the actual duration may be longer. Also, if the requested duration is too large for the radio to accommodate, [RAIL_StartCcaLbtTx\(\)](#) will fail returning [RAIL_STATUS_INVALID_PARAMETER](#).

Definition at line 3545 of file `common/rail_types.h`

lbtTimeout

```
RAIL_Time_t RAIL_LbtConfig_t::lbtTimeout
```

An overall timeout, in RAIL's microsecond time base, for the operation.

If the transmission doesn't start before this timeout expires, the transmission will fail with [RAIL_EVENT_TX_CHANNEL_BUSY](#). This is important for limiting LBT due to LBT's unbounded requirement that if the channel is busy, the next try must wait for the channel to clear. A value 0 means no timeout is imposed.

Definition at line 3554 of file `common/rail_types.h`

RAIL_SyncWordConfig_t

RAIL sync words and length configuration.

Public Attributes

uint8_t	syncWordBits	Sync word length in bits, between 2 and 32, inclusive.
uint32_t	syncWord1	Sync Word1.
uint32_t	syncWord2	Sync Word2.

Public Attribute Documentation

syncWordBits

uint8_t RAIL_SyncWordConfig_t::syncWordBits

Sync word length in bits, between 2 and 32, inclusive.

Definition at line 3601 of file common/rail_types.h

syncWord1

uint32_t RAIL_SyncWordConfig_t::syncWord1

Sync Word1.

Note

- Only the least-significant [syncWordBits](#) bits are used, which are sent or received on air least-significant-bit first.

Definition at line 3607 of file common/rail_types.h

syncWord2

uint32_t RAIL_SyncWordConfig_t::syncWord2

Sync Word2.

Note

- Only the least-significant [syncWordBits](#) bits are used, which are sent or received on air least-significant-bit first.

Definition at line 3613 of file common/rail_types.h

RAIL_TxRepeatConfig_t

A configuration structure for repeated transmits.

Note

- The PA will always be ramped down and up in between transmits so there will always be some minimum delay between transmits depending on the ramp time configuration.

Public Attributes

uint16_t	iterations	The number of repeated transmits to run.
RAIL_TxRepeatOptions_t	repeatOptions	Repeat option(s) to apply.
RAIL_TransitionTime_t	delay	When RAIL_TX_REPEAT_OPTION_HOP is not set, specifies the delay time between each repeated transmit.
RAIL_TxChannelHoppingConfig_t	channelHopping	When RAIL_TX_REPEAT_OPTION_HOP is set, this specifies the channel hopping configuration to use when hopping between repeated transmits.
union RAIL_TxRepeatConfig_t::@0	delayOrHop	Per-repeat delay or hopping configuration, depending on repeatOptions.

Public Attribute Documentation

iterations

```
uint16_t RAIL_TxRepeatConfig_t::iterations
```

The number of repeated transmits to run.

A total of (iterations + 1) transmits will go on-air in the absence of errors.

Definition at line 3654 of file common/rail_types.h

repeatOptions

```
RAIL_TxRepeatOptions_t RAIL_TxRepeatConfig_t::repeatOptions
```

Repeat option(s) to apply.

Definition at line 3658 of file common/rail_types.h

delay

```
RAIL_TransitionTime_t RAIL_TxRepeatConfig_t::delay
```

When [RAIL_TX_REPEAT_OPTION_HOP](#) is not set, specifies the delay time between each repeated transmit.

Specify [RAIL_TRANSITION_TIME_KEEP](#) to use the current [RAIL_StateTiming_t::txToTx](#) transition time setting. When using [RAIL_TX_REPEAT_OPTION_START_TO_START](#) the delay must be long enough to cover the prior transmit's time.

Definition at line 3671 of file `common/rail_types.h`

channelHopping

```
RAIL_TxChannelHoppingConfig_t RAIL_TxRepeatConfig_t::channelHopping
```

When [RAIL_TX_REPEAT_OPTION_HOP](#) is set, this specifies the channel hopping configuration to use when hopping between repeated transmits.

Per-hop delays are configured within each [RAIL_TxChannelHoppingConfigEntry_t::delay](#) rather than this union's delay field. When using [RAIL_TX_REPEAT_OPTION_START_TO_START](#) the hop delay must be long enough to cover the prior transmit's time.

Definition at line 3681 of file `common/rail_types.h`

delayOrHop

```
union RAIL_TxRepeatConfig_t::@0 RAIL_TxRepeatConfig_t::delayOrHop
```

Per-repeat delay or hopping configuration, depending on repeatOptions.

Definition at line 3682 of file `common/rail_types.h`

Packet Transmit

Packet Transmit

APIs which initiate a packet transmission in RAIL.

When using any of these functions, the data to be transmitted must have been previously written to the transmit FIFO via [RAIL_SetTxFifo\(\)](#) and/or [RAIL_WriteTxFifo\(\)](#).

Functions

RAIL_Status_t	RAIL_StartTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_SchedulerInfo_t *schedulerInfo) Start a transmit.
RAIL_Status_t	RAIL_StartScheduledTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_ScheduleTxConfig_t *config, const RAIL_SchedulerInfo_t *schedulerInfo) Schedule sending a packet.
RAIL_Status_t	RAIL_StartCcaCsmaTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_CsmaConfig_t *csmaConfig, const RAIL_SchedulerInfo_t *schedulerInfo) Start a transmit using CSMA.
RAIL_Status_t	RAIL_StartCcaLbtTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_LbtConfig_t *lbtConfig, const RAIL_SchedulerInfo_t *schedulerInfo) Start a transmit using LBT.
RAIL_Status_t	RAIL_StartScheduledCcaCsmaTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_ScheduleTxConfig_t *scheduleTxConfig, const RAIL_CsmaConfig_t *csmaConfig, const RAIL_SchedulerInfo_t *schedulerInfo) Schedule a transmit using CSMA.
RAIL_Status_t	RAIL_StartScheduledCcaLbtTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_ScheduleTxConfig_t *scheduleTxConfig, const RAIL_LbtConfig_t *lbtConfig, const RAIL_SchedulerInfo_t *schedulerInfo) Schedule a transmit using LBT.

Function Documentation

RAIL_StartTx

RAIL_Status_t RAIL_StartTx ([RAIL_Handle_t](#) railHandle, uint16_t channel, [RAIL_TxOptions_t](#) options, const [RAIL_SchedulerInfo_t](#) * schedulerInfo)

Start a transmit.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	Define the channel to transmit on.
[in]	options	TX options to be applied to this transmit only.
[in]	schedulerInfo	A pointer to information to allow the radio scheduler to place this transmit appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

Status code indicating success of the function call. If successfully initiated, transmit completion or failure will be reported by a later [RAIL_Config_t::eventsCallback](#) with the appropriate [RAIL_Events_t](#).

The transmit process will begin immediately or as soon as a packet being received has finished. The data to be transmitted must have been previously established via [RAIL_SetTxFifo\(\)](#) and/or [RAIL_WriteTxFifo\(\)](#).

Returns an error if a previous transmit is still in progress. If changing channels, any ongoing packet reception is aborted.

In multiprotocol, ensure that the radio is properly yielded after this operation completes. See [Yielding the Radio](#) for more details.

Definition at line 3203 of file `common/rail.h`

RAIL_StartScheduledTx

```
RAIL_Status_t RAIL_StartScheduledTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_ScheduleTxConfig_t * config, const RAIL_SchedulerInfo_t * schedulerInfo)
```

Schedule sending a packet.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	The channel to transmit on.
[in]	options	TX options to be applied to this transmit only.
[in]	config	A pointer to the RAIL_ScheduleTxConfig_t structure indicating when and how the transmit should occur.
[in]	schedulerInfo	A pointer to information to allow the radio scheduler to place this transmit appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call. If successfully initiated, a transmit completion or failure will be reported by a later [RAIL_Config_t::eventsCallback](#) with the appropriate [RAIL_Events_t](#).

The transmit process will begin at the scheduled time. The data to be transmitted must have been previously established via [RAIL_SetTxFifo\(\)](#) and/or [RAIL_WriteTxFifo\(\)](#). The time (in microseconds) and whether that time is absolute or relative is specified using the [RAIL_ScheduleTxConfig_t](#) structure. What to do if a scheduled transmit fires in the middle of receiving a packet is also specified in this structure.

Returns an error if a previous transmit is still in progress. If changing channels, the channel is changed immediately and will abort any ongoing packet reception.

In multiprotocol, ensure that the radio is properly yielded after this operation completes. See [Yielding the Radio](#) for more details.

Definition at line 3238 of file `common/rail.h`

RAIL_StartCcaCsmaTx

```
RAIL_Status_t RAIL_StartCcaCsmaTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_CsmaConfig_t * csmaConfig, const RAIL_SchedulerInfo_t * schedulerInfo)
```

Start a transmit using CSMA.

Parameters

[in]	railHandle	A RAIL instance handle.

[in]	channel	The channel to transmit on.
[in]	options	TX options to be applied to this transmit only.
[in]	csmaConfig	A pointer to the RAIL_CsmaConfig_t structure describing the CSMA parameters to use for this transmit. In multiprotocol this must point to global or heap storage that remains valid after the API returns until the transmit is actually started.
[in]	schedulerInfo	A pointer to information to allow the radio scheduler to place this transmit appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call. If successfully initiated, a transmit completion or failure will be reported by a later [RAIL_Config_t::eventsCallback](#) with the appropriate [RAIL_Events_t](#).

Perform the Carrier Sense Multiple Access (CSMA) algorithm, and if the channel is deemed clear (RSSI below the specified threshold), it will commence transmission. The data to be transmitted must have been previously established via [RAIL_SetTxFifo\(\)](#) and/or [RAIL_WriteTxFifo\(\)](#). Packets can be received during CSMA backoff periods if receive is active throughout the CSMA process. This will happen either by starting the CSMA process while receive is already active, or if the [RAIL_CsmaConfig_t::ccaBackoff](#) time is less than the [RAIL_StateTiming_t::idleToRx](#) time. If the [RAIL_CsmaConfig_t::ccaBackoff](#) time is greater than the [RAIL_StateTiming_t::idleToRx](#) time, receive will only be active during CSMA's clear channel assessments.

If the CSMA algorithm deems the channel busy, the [RAIL_Config_t::eventsCallback](#) occurs with [RAIL_EVENT_TX_CHANNEL_BUSY](#), and the contents of the transmit FIFO remain intact.

Returns an error if a previous transmit is still in progress. If changing channels, the channel is changed immediately and any ongoing packet reception is aborted.

Returns an error if a scheduled RX is still in progress.

In multiprotocol, ensure that the radio is properly yielded after this operation completes. See [Yielding the Radio](#) for more details.

Definition at line 3286 of file `common/rail.h`

RAIL_StartCcaLbtTx

```
RAIL_Status_t RAIL_StartCcaLbtTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_LbtConfig_t * lbtConfig, const RAIL_SchedulerInfo_t * schedulerInfo)
```

Start a transmit using LBT.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	The channel to transmit on.
[in]	options	TX options to be applied to this transmit only.
[in]	lbtConfig	A pointer to the RAIL_LbtConfig_t structure describing the LBT parameters to use for this transmit. In multiprotocol this must point to global or heap storage that remains valid after the API returns until the transmit is actually started.
[in]	schedulerInfo	A pointer to information to allow the radio scheduler to place this transmit appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call. If successfully initiated, a transmit completion or failure will be reported by a later [RAIL_Config_t::eventsCallback](#) with the appropriate [RAIL_Events_t](#).

Performs the Listen Before Talk (LBT) algorithm, and if the channel is deemed clear (RSSI below the specified threshold), it will commence transmission. The data to be transmitted must have been previously established via [RAIL_SetTxFifo\(\)](#) and/or [RAIL_WriteTxFifo\(\)](#). Packets can be received during LBT backoff periods if receive is active throughout the LBT process. This will happen either by starting the LBT process while receive is already active, or if the [RAIL_LbtConfig_t::lbtBackoff](#) time is less than the [RAIL_StateTiming_t::idleToRx](#) time. If the [RAIL_LbtConfig_t::lbtBackoff](#) time is greater than the [RAIL_StateTiming_t::idleToRx](#) time, receive will only be active during LBT's clear channel assessments.

If the LBT algorithm deems the channel busy, the [RAIL_Config_t::eventsCallback](#) occurs with [RAIL_EVENT_TX_CHANNEL_BUSY](#), and the contents of the transmit FIFO remain intact.

Returns an error if a previous transmit is still in progress. If changing channels, the channel is changed immediately and any ongoing packet reception is aborted.

Returns an error if a scheduled RX is still in progress.

In multiprotocol, ensure that the radio is properly yielded after this operation completes. See [Yielding the Radio](#) for more details.

Definition at line 3334 of file `common/rail.h`

RAIL_StartScheduledCcaCsmaTx

```
RAIL_Status_t RAIL_StartScheduledCcaCsmaTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options,
const RAIL_ScheduleTxConfig_t * scheduleTxConfig, const RAIL_CsmaConfig_t * csmaConfig, const RAIL_SchedulerInfo_t *
schedulerInfo)
```

Schedule a transmit using CSMA.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	The channel to transmit on.
[in]	options	TX options to be applied to this transmit only.
[in]	scheduleTxConfig	A pointer to the RAIL_ScheduleTxConfig_t structure describing the CSMA parameters to use for this transmit.
[in]	csmaConfig	A pointer to the RAIL_CsmaConfig_t structure describing the CSMA parameters to use for this transmit. In multiprotocol this must point to global or heap storage that remains valid after the API returns until the transmit is actually started.
[in]	schedulerInfo	A pointer to information to allow the radio scheduler to place this transmit appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call. If successfully initiated, a transmit completion or failure will be reported by a later [RAIL_Config_t::eventsCallback](#) with the appropriate [RAIL_Events_t](#).

Internally, the RAIL library needs a PRS channel for this feature. It will allocate an available PRS channel to use and hold onto that channel for future use. If no PRS channel is available, the function returns with [RAIL_STATUS_INVALID_CALL](#).

This is a scheduled variation of [RAIL_StartCcaCsmaTx\(\)](#); refer to that function for CSMA details.

Returns an error if a previous transmit is still in progress. If changing channels, the channel is changed immediately and any ongoing packet reception is aborted.

Returns an error if a scheduled RX is still in progress.

In multiprotocol, ensure that the radio is properly yielded after this operation completes. See [Yielding the Radio](#) for more details.

Definition at line 3376 of file common/rail.h

RAIL_StartScheduledCcaLbtTx

```
RAIL_Status_t RAIL_StartScheduledCcaLbtTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_ScheduleTxConfig_t * scheduleTxConfig, const RAIL_LbtConfig_t * lbtConfig, const RAIL_SchedulerInfo_t * schedulerInfo)
```

Schedule a transmit using LBT.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	The channel to transmit on.
[in]	options	TX options to be applied to this transmit only.
[in]	scheduleTxConfig	A pointer to the RAIL_ScheduleTxConfig_t structure describing the CSMA parameters to use for this transmit.
[in]	lbtConfig	A pointer to the RAIL_LbtConfig_t structure describing the LBT parameters to use for this transmit. In multiprotocol this must point to global or heap storage that remains valid after the API returns until the transmit is actually started.
[in]	schedulerInfo	A pointer to information to allow the radio scheduler to place this transmit appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call. If successfully initiated, a transmit completion or failure will be reported by a later RAIL_Config_t::eventsCallback with the appropriate RAIL_Events_t.

Internally, the RAIL library needs a PRS channel for this feature. It will allocate an available PRS channel to use and hold onto that channel for future use. If no PRS channel is available, the function returns with RAIL_STATUS_INVALID_CALL.

This is a scheduled variation of RAIL_StartCcaLbtTx; refer to that function for LBT details.

Returns an error if a previous transmit is still in progress. If changing channels, the channel is changed immediately and any ongoing packet reception is aborted.

Returns an error if a scheduled RX is still in progress.

In multiprotocol, ensure that the radio is properly yielded after this operation completes. See Yielding the Radio for more details.

Definition at line 3419 of file common/rail.h

Power Amplifier (PA)

Power Amplifier (PA)

APIs for interacting with one of the on chip PAs.

These APIs let you configure the on-chip PA to get the appropriate output power.

These are the function types: 1) Configuration functions: These functions set and get configuration for the PA. In this case, "configuration" refers to a) indicating which PA to use, b) the voltage supplied by your board to the PA, and c) the ramp time over which to ramp the PA up to its full power. 2) Power-setting functions: These functions consume the actual values written to the PA registers and write them appropriately. These values are referred to as "(raw) power levels". The range of acceptable values for these functions depends on which PA is currently active. The higher the power level set, the higher the dBm power output by the radio. However, the mapping between dBm and these power levels can vary greatly between modules/boards. 3) Conversion functions: These functions convert between the "power levels" discussed previously and the dBm values output by the radio. Continue reading for more information about unit conversion.

The accuracy of the radio output power is application-specific. For some protocols or channels, the protocol itself or legal limitations require applications to know exactly what power they're transmitting at, in dBm. Other applications do not have these restrictions, and users determine power level(s) that fit their criteria for the trade-off between radio range and power savings, regardless of what dBm power that maps to.

[RAIL_ConvertRawToDbm\(\)](#) and [RAIL_ConvertDbmToRaw\(\)](#), which convert between the dBm power and the raw power levels, provide a solution that fits all these applications. The levels of customization are outlined below: 1) No customization needed: for a given dBm value, the result of [RAIL_ConvertDbmToRaw\(\)](#) provides an appropriate raw power level that, when written to the registers via [RAIL_SetTxPower\(\)](#), causes the radio to output at that dBm power. In this case, no action is needed by the user, the WEAK versions of the conversion functions can be used and the default include paths in [pa_conversions_efr32.h](#) can be used. 2) The mapping of power level to dBm is not ideal, but the level of precision is sufficient: In [pa_conversions_efr32.c](#), the WEAK versions of the conversion functions work by using 8-segment piecewise linear curves to convert between dBm and power levels for PA's with hundreds of power levels and simple mapping tables for use with PA's with only a few levels. If this method is sufficiently precise, but the mapping between power levels and dBm is incorrect, copy [pa_curves_efr32.h](#) into a new file, updating the segments to form a better fit ([_DCDC_CURVES](#) or [_VBAT_CURVES](#) defines) and then add the [RAIL_PA_CURVES](#) define to your build with the path to the new file. 3) A different level of precision is needed and the fit is bad: If the piecewise-linear line segment fit is not appropriate for your solution, the functions in [pa_conversions_efr32.c](#) can be totally rewritten, as long as [RAIL_ConvertDbmToRaw\(\)](#) and [RAIL_ConvertRawToDbm\(\)](#) have the same signatures. It is completely acceptable to re-write these in a way that makes the [pa_curves_efr32.h](#) and [pa_curve_types_efr32.h](#) files referenced in [pa_conversions_efr32.h](#) unnecessary. Those files are needed solely for the provided conversion methods. 4) dBm values are not necessary: If the application does not require dBm values at all, overwrite [RAIL_ConvertDbmToRaw\(\)](#) and [RAIL_ConvertRawToDbm\(\)](#) with smaller functions (i.e., return 0 or whatever was input). These functions are called from within the RAIL library, so they can never be deadstripped, but making them as small as possible is the best way to reduce code size. From there, call [RAIL_SetTxPower\(\)](#), without converting from a dBm value. To stop the library from coercing the power based on channels, overwrite [RAIL_ConvertRawToDbm\(\)](#) to always return 0 and overwrite [RAIL_ConvertDbmToRaw\(\)](#) to always return 255.

The following is example code that shows how to initialize your PA

```
#include "pa_conversions_efr32.h"

// A macro RAIL_TX_POWER_CURVES_CONFIG is used as the curve
// structures used by the provided conversion functions.
RAIL_TxPowerCurvesConfigAlt_t tx_power_curves_config = RAIL_TX_POWER_CURVES_CONFIG;
// Saves those curves
// to be referenced when the conversion functions are called.
RAIL_InitTxPowerCurves(rail_handle, &tx_power_curves_config);

// Declares the structure used to configure the PA.
RAIL_TxPowerConfig_t txPowerConfig = {
    .mode    = RAIL_TX_POWER_MODE_2P4_HP,
    .voltage = 3300,
    .rampTime = 10,
};

// Initializes the PA. Here, it is assumed that 'railHandle' is a valid RAIL_Handle_t
// that has already been initialized.
RAIL_ConfigTxPower(railHandle, &txPowerConfig);

// Picks a dBm power to use: 100 deci-dBm = 10 dBm. See docs on RAIL_TxPower_t.
RAIL_TxPower_t power = 100;

// Gets the config written by RAIL_ConfigTxPower() to confirm what was actually set.
RAIL_GetTxPowerConfig(railHandle, &txPowerConfig);

// RAIL_ConvertDbmToRaw() is the default weak version,
// or the customer version, if overwritten.
RAIL_TxPowerLevel_t powerLevel
    = RAIL_ConvertDbmToRaw(railHandle, txPowerConfig.mode, power);

// Writes the result of the conversion to the PA power registers in terms
// of raw power levels.
RAIL_SetTxPower(railHandle, powerLevel);
```

Note

- All lines following "RAIL_TxPower_t power = 100;" can be replaced with the provided utility function, [RAIL_SetTxPowerDbm\(\)](#). However, the full example here was provided for clarity. See the documentation on [RAIL_SetTxPowerDbm\(\)](#) for more details.

Modules

[RAIL_TxPowerSettingEntry_t](#)

[RAIL_TxPowerConfig_t](#)

[RAIL_PaAutoModeConfigEntry_t](#)

[sl_rail_nvm_pa_curve_t](#)

[sl_rail_nvm_pa_table_t](#)

[sl_rail_nvm_pa_descriptor_t](#)

[sl_rail_nvm_pa_config_t](#)

[EFR32xG2x](#)

[PA Curve Conversions](#)

[SIxx3xx](#)

Enumerations

```
enum    RAIL_TxPowerMode_t {
```

```

RAIL_TX_POWER_MODE_2P4GIG_HP = 0U
RAIL_TX_POWER_MODE_2P4_HP =
RAIL_TX_POWER_MODE_2P4GIG_HP
RAIL_TX_POWER_MODE_2P4GIG_MP = 1U
RAIL_TX_POWER_MODE_2P4_MP =
RAIL_TX_POWER_MODE_2P4GIG_MP
RAIL_TX_POWER_MODE_2P4GIG_LP = 2U
RAIL_TX_POWER_MODE_2P4_LP =
RAIL_TX_POWER_MODE_2P4GIG_LP
RAIL_TX_POWER_MODE_2P4GIG_LLP = 3U
RAIL_TX_POWER_MODE_2P4GIG_HIGHEST = 4U
RAIL_TX_POWER_MODE_2P4_HIGHEST =
RAIL_TX_POWER_MODE_2P4GIG_HIGHEST
RAIL_TX_POWER_MODE_SUBGIG_POWERSETTING_TABLE
= 5U
RAIL_TX_POWER_MODE_SUBGIG_HP = 6U
RAIL_TX_POWER_MODE_SUBGIG =
RAIL_TX_POWER_MODE_SUBGIG_HP
RAIL_TX_POWER_MODE_SUBGIG_MP = 7U
RAIL_TX_POWER_MODE_SUBGIG_LP = 8U
RAIL_TX_POWER_MODE_SUBGIG_LLP = 9U
RAIL_TX_POWER_MODE_SUBGIG_HIGHEST = 10U
RAIL_TX_POWER_MODE_OFDM_PA_POWERSETTING_TABLE
= 11U
RAIL_TX_POWER_MODE_OFDM_PA =
RAIL_TX_POWER_MODE_OFDM_PA_POWERSETTING_TABLE
RAIL_TX_POWER_MODE_NONE
}

```

An enumeration of the EFR32 power modes.

```

enum    RAIL_PaBand_t {
        RAIL_PA_BAND_2P4GIG
        RAIL_PA_BAND_SUBGIG
        RAIL_PA_BAND_COUNT
    }

```

Enum used to specify the band for a PA.

Typedefs

typedef int16_t	RAIL_TxPower_t The transmit power in deci-dBm units (e.g., 4.5 dBm -> 45 deci-dBm).
typedef uint8_t	RAIL_TxPowerLevel_t Raw power levels used directly by RAIL_GetTxPower() and RAIL_SetTxPower() where a higher numerical value corresponds to a higher output power.
typedef uint32_t	RAIL_PaPowerSetting_t PA power setting used directly by RAIL_GetPaPowerSetting() and RAIL_SetPaPowerSetting() which is decoded to the actual hardware register value(s).

Variables

const RAIL_PaAutoModeConfigEntry_t *	RAIL_PaAutoModeConfig The actual PA auto mode configuration structure used by the auto mode plugin to control output power.

Functions

RAIL_Status_t	RAIL_ConfigTxPower (RAIL_Handle_t railHandle, const RAIL_TxPowerConfig_t *config) Initialize TX power settings.
RAIL_Status_t	RAIL_GetTxPowerConfig (RAIL_Handle_t railHandle, RAIL_TxPowerConfig_t *config)

Get the TX power settings currently used in the amplifier.

RAIL_Status_t	RAIL_SetTxPower (RAIL_Handle_t railHandle, RAIL_TxPowerLevel_t powerLevel) Set the TX power in units of raw units (see rail_chip_specific.h for value ranges).
RAIL_TxPowerLevel_t	RAIL_GetTxPower (RAIL_Handle_t railHandle) Return the current power setting of the PA.
RAIL_TxPower_t	RAIL_ConvertRawToDbm (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPowerLevel_t powerLevel) Convert raw values written to registers to decibel value (in units of deci-dBm).
RAIL_TxPowerLevel_t	RAIL_ConvertDbmToRaw (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPower_t power) Convert the desired decibel value (in units of deci-dBm) to raw integer values used by the TX amplifier registers.
RAIL_Status_t	RAIL_ConvertDbmToPowerSettingEntry (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPower_t power, RAIL_TxPowerSettingEntry_t *pPowerSettingInfo) Converts the desired decibel value (in units of deci-dBm) to a RAIL_TxPowerSettingEntry_t .
RAIL_Status_t	RAIL_VerifyTxPowerCurves (const struct RAIL_TxPowerCurvesConfigAlt *config) Verify the TX Power Curves on modules.
RAIL_Status_t	RAIL_SetTxPowerDbm (RAIL_Handle_t railHandle, RAIL_TxPower_t power) Set the TX power in terms of deci-dBm instead of raw power level.
RAIL_TxPower_t	RAIL_GetTxPowerDbm (RAIL_Handle_t railHandle) Get the TX power in terms of deci-dBm instead of raw power level.
const RAIL_PaPowerSetting_t *	RAIL_GetPowerSettingTable (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPower_t *minPower, RAIL_TxPower_t *maxPower, RAIL_TxPowerLevel_t *step) Get the TX PA power setting table and related values.
RAIL_Status_t	RAIL_SetPaPowerSetting (RAIL_Handle_t railHandle, RAIL_PaPowerSetting_t paPowerSetting, RAIL_TxPower_t minPowerDbm, RAIL_TxPower_t maxPowerDbm, RAIL_TxPower_t currentPowerDbm) Set the TX PA power setting used to configure the PA hardware for the PA output power determined by RAIL_SetTxPowerDbm() .
RAIL_PaPowerSetting_t	RAIL_GetPaPowerSetting (RAIL_Handle_t railHandle) Get the TX PA power setting, which is used to configure power configurations when the dBm to paPowerSetting mapping table mode is used.
RAIL_Status_t	RAIL_EnablePaAutoMode (RAIL_Handle_t railHandle, bool enable) Enable automatic switching between PAs internally to the RAIL library.
bool	RAIL_IsPaAutoModeEnabled (RAIL_Handle_t railHandle) Query status of PA Auto Mode.
RAIL_Status_t	RAILCb_PaAutoModeDecision (RAIL_Handle_t railHandle, RAIL_TxPower_t *power, RAIL_TxPowerMode_t *mode, const RAIL_ChannelConfigEntry_t *chCfgEntry) Callback that decides which PA and power level should be used while in PA auto mode.
RAIL_Status_t	RAIL_ConfigPaAutoEntry (RAIL_Handle_t railHandle, const RAIL_PaAutoModeConfigEntry_t *paAutoModeEntry) Configure the PA auto mode entries.
sl_status_t	sl_rail_util_pa_nvm_write_mode (RAIL_TxPowerMode_t pa_mode) Store a RAIL_TxPowerMode_t into NVM.
RAIL_TxPowerMode_t	sl_rail_util_pa_nvm_read_mode (void) Reads stored RAIL_TxPowerMode_t from NVM.
sl_status_t	sl_rail_util_pa_nvm_write_config (const sl_rail_nvm_pa_config_t *p_pa_config) Stores a sl_rail_nvm_pa_config_t into NVM.

sl_status_t [sl_rail_util_pa_nvm_read_config](#)(sl_rail_nvm_pa_config_t *p_pa_config)
Reads stored [sl_rail_nvm_pa_config_t](#) from NVM.

Macros

#define	RAIL_TX_POWER_MAX ((RAIL_TxPower_t)0x7FFF) The maximum valid value for a RAIL_TxPower_t .
#define	RAIL_TX_POWER_MIN ((RAIL_TxPower_t)0x8000) The minimum valid value for a RAIL_TxPower_t .
#define	RAIL_TX_POWER_CURVE_DEFAULT_MAX ((RAIL_TxPower_t)200) The maximum power in deci-dBm the curve supports.
#define	RAIL_TX_POWER_CURVE_DEFAULT_INCREMENT ((RAIL_TxPower_t)40) The increment step in deci-dBm for calculating power level.
#define	RAIL_TX_POWER_VOLTAGE_SCALING_FACTOR 1000 mV are used for all TX power voltage values.
#define	RAIL_TX_POWER_DBM_SCALING_FACTOR 10 deci-dBm are used for all TX power dBm values.
#define	RAIL_TX_POWER_LEVEL_INVALID (255U) Invalid RAIL_TxPowerLevel_t value returned when an error occurs with RAIL_GetTxPower() .
#define	RAIL_TX_POWER_LEVEL_MAX (254U) Sentinel value that can be passed to RAIL_SetTxPower() to set the highest power level available on the current PA, regardless of which one is selected.
#define	RAIL_TX_PA_POWER_SETTING_UNSUPPORTED (0U) Returned by RAIL_GetPaPowerSetting() when the device does not support the dBm to power setting mapping table.
#define	RAIL_TX_POWER_MODE_NAMES undefined The names of the TX power modes.
#define	RAIL_POWER_MODE_IS_DBM_POWERSETTING_MAPPING_TABLE_OFDM (x) Convenience macro for any OFDM mapping table mode.
#define	RAIL_POWER_MODE_IS_DBM_POWERSETTING_MAPPING_TABLE_SUBGIG (x) Convenience macro for any Sub-GHz mapping table mode.
#define	RAIL_POWER_MODE_IS_ANY_OFDM (x) Convenience macro for any OFDM mode.
#define	SL_RAIL_UTIL_PA_NVM_MODE_TAG (0x89881UL) The NVM3 PA mode tag.
#define	SL_RAIL_UTIL_PA_NVM_CONFIG_TAG (0x89882UL) The NVM3 PA config tag.
#define	SL_RAIL_NVM_PA_CONFIGS_VERSION 1 The version number of the NVM structures defined herein.
#define	SL_RAIL_NVM_PA_COUNT RAIL_NUM_PA The number of curves/tables allowed in NVM storage.
#define	SL_RAIL_NVM_PA_CURVE_SEGMENTS 9 The number of curve segments allowed per curve in NVM storage.
#define	SL_RAIL_NVM_PA_TABLE_ENTRIES 16 The number of table values allowed per mapping table in NVM storage.

Enumeration Documentation

RAIL_TxPowerMode_t

RAIL_TxPowerMode_t

An enumeration of the EFR32 power modes.

The power modes on the EFR32 correspond to the different on-chip PAs that are available. For more information about the power and performance characteristics of a given amplifier, see the data sheet.

Enumerator	
RAIL_TX_POWER_MODE_2P4GIG_HP	High-power 2.4 GHz amplifier EFR32xG21: up to 20 dBm, raw values: 1-180 EFR32xG22: up to 6 dBm, raw values: 1-128 EFR32xG24: up to 20 dBm, raw values: 0-180, or up to 10 dBm, raw values: 0-90 EFR32xG26: same as EFR32xG24 EFR32xG27: up to 6 dBm, raw values: 1-128 EFR32xG28: up to 10 dBm, raw values: 0-240 Not supported on other platforms.
RAIL_TX_POWER_MODE_2P4_HP	Deprecated Please use RAIL_TX_POWER_MODE_2P4GIG_HP instead.
RAIL_TX_POWER_MODE_2P4GIG_MP	Mid-power 2.4 GHz amplifier EFR32xG21: up to 10 dBm, raw values: 1-90 Not supported on other platforms.
RAIL_TX_POWER_MODE_2P4_MP	Deprecated Please use RAIL_TX_POWER_MODE_2P4GIG_MP instead.
RAIL_TX_POWER_MODE_2P4GIG_LP	Low-power 2.4 GHz amplifier EFR32xG21: up to 0 dBm, raw values: 1-64 EFR32xG22: up to 0 dBm, raw values: 1-16 EFR32xG24: up to 0 dBm, raw values: 1-16 EFR32xG26: same as EFR32xG24 EFR32xG27: up to 0 dBm, raw values: 1-16 Not supported on other platforms.
RAIL_TX_POWER_MODE_2P4_LP	Deprecated Please use RAIL_TX_POWER_MODE_2P4GIG_LP instead.
RAIL_TX_POWER_MODE_2P4GIG_LLP	Low-Low-power 2.4 GHz amplifier Not currently supported on any EFR32 platform.
RAIL_TX_POWER_MODE_2P4GIG_HIGHEST	Select the highest 2.4 GHz power PA available on the current chip.
RAIL_TX_POWER_MODE_2P4_HIGHEST	Deprecated Please use RAIL_TX_POWER_MODE_2P4GIG_HIGHEST instead.
RAIL_TX_POWER_MODE_SUBGIG_POWERSETTING_TABLE	PA for all Sub-GHz dBm values in range, using RAIL_PaPowerSetting_t table.
RAIL_TX_POWER_MODE_SUBGIG_HP	High-power Sub-GHz amplifier (Class D mode) Supported on EFR32xG23 and EFR32xG28.
RAIL_TX_POWER_MODE_SUBGIG	Deprecated Please use RAIL_TX_POWER_MODE_SUBGIG_HP instead.
RAIL_TX_POWER_MODE_SUBGIG_MP	Mid-power Sub-GHz amplifier Supported only on EFR32xG23 and EFR32xG28.
RAIL_TX_POWER_MODE_SUBGIG_LP	Low-power Sub-GHz amplifier Supported only on EFR32xG23 and EFR32xG28.
RAIL_TX_POWER_MODE_SUBGIG_LLP	Low-Low-power Sub-GHz amplifier Supported only on EFR32xG23 and EFR32xG28.
RAIL_TX_POWER_MODE_SUBGIG_HIGHEST	Select the highest Sub-GHz power PA available on the current chip.

RAIL_TX_POWER_MODE_OFDM_PA_POWERSETTING_TABLE	PA for all OFDM Sub-GHz dBm values in range, using RAIL_PaPowerSetting_t table.
RAIL_TX_POWER_MODE_OFDM_PA	Deprecated Please use RAIL_TX_POWER_MODE_OFDM_PA_POWERSETTING_TABLE instead.
RAIL_TX_POWER_MODE_NONE	Invalid amplifier Selection.

Definition at line 1801 of file `common/rail_types.h`

RAIL_PaBand_t

RAIL_PaBand_t

Enum used to specify the band for a PA.

Enumerator	
RAIL_PA_BAND_2P4GIG	Indicates a 2.4 GHz band PA.
RAIL_PA_BAND_SUBGIG	Indicates a Sub-GHz band PA.
RAIL_PA_BAND_COUNT	A count of the choices in this enumeration.

Definition at line 51 of file `plugin/pa-auto-mode/pa_auto_mode.h`

Typedef Documentation

RAIL_TxPower_t

typedef int16_t RAIL_TxPower_t

The transmit power in deci-dBm units (e.g., 4.5 dBm -> 45 deci-dBm).

These values are used by the conversion functions to convert a [RAIL_TxPowerLevel_t](#) to deci-dBm for the application consumption. On EFR32, they can range from [RAIL_TX_POWER_MIN](#) to [RAIL_TX_POWER_MAX](#).

Definition at line 1719 of file `common/rail_types.h`

RAIL_TxPowerLevel_t

typedef uint8_t RAIL_TxPowerLevel_t

Raw power levels used directly by [RAIL_GetTxPower\(\)](#) and [RAIL_SetTxPower\(\)](#) where a higher numerical value corresponds to a higher output power.

These are referred to as 'raw (values/units)'. On EFR32, they can range from one of [RAIL_TX_POWER_LEVEL_2P4_LP_MIN](#), [RAIL_TX_POWER_LEVEL_2P4_HP_MIN](#), or [RAIL_TX_POWER_LEVEL_SUBGIG_HP_MIN](#) to one of [RAIL_TX_POWER_LEVEL_2P4_LP_MAX](#), [RAIL_TX_POWER_LEVEL_2P4_HP_MAX](#), and [RAIL_TX_POWER_LEVEL_SUBGIG_HP_MAX](#), respectively, depending on the selected [RAIL_TxPowerMode_t](#).

Definition at line 1749 of file `common/rail_types.h`

RAIL_PaPowerSetting_t

```
typedef uint32_t RAIL_PaPowerSetting_t
```

PA power setting used directly by [RAIL_GetPaPowerSetting\(\)](#) and [RAIL_SetPaPowerSetting\(\)](#) which is decoded to the actual hardware register value(s).

Definition at line 1769 of file `common/rail_types.h`

Variable Documentation

RAIL_PaAutoModeConfig

```
const RAIL_PaAutoModeConfigEntry_t* RAIL_PaAutoModeConfig
```

The actual PA auto mode configuration structure used by the auto mode plugin to control output power.

Definition at line 87 of file `plugin/pa-auto-mode/pa_auto_mode.h`

Function Documentation

RAIL_ConfigTxPower

```
RAIL_Status_t RAIL_ConfigTxPower (RAIL\_Handle\_t railHandle, const RAIL\_TxPowerConfig\_t * config)
```

Initialize TX power settings.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	A pointer to a power config with the desired initial settings for the TX amplifier.

Returns

- Status code indicating success of the function call.

These settings include the selection between the multiple TX amplifiers, voltage supplied to the TX power amplifier, and ramp times. This must be called before any transmit occurs or [RAIL_SetTxPower\(\)](#) is called. While this function should always be called during initialization, it can also be called any time if these settings need to change to adapt to a different application/protocol. This API also resets TX power to [RAIL_TX_POWER_LEVEL_INVALID](#), so [RAIL_SetTxPower\(\)](#) must be called afterwards.

At times, certain combinations of configurations cannot be achieved. This API attempts to get as close as possible to the requested settings. The following "RAIL_GetTxPower..." API can be used to determine what values were set. A change in [RAIL_TxPowerConfig_t::rampTime](#) may affect the minimum timings that can be achieved in [RAIL_StateTiming_t::idleToTx](#) and [RAIL_StateTiming_t::rxToTx](#). Call [RAIL_SetStateTiming\(\)](#) again to check whether these times have changed.

Definition at line 2840 of file `common/rail.h`

RAIL_GetTxPowerConfig

```
RAIL_Status_t RAIL_GetTxPowerConfig (RAIL\_Handle\_t railHandle, RAIL\_TxPowerConfig\_t * config)
```

Get the TX power settings currently used in the amplifier.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	config	A non-NULL pointer to a RAIL_TxPowerConfig_t structure filled in by the function.

Returns

- Status code indicating success of the function call.

Note that this API does not return the current TX power, which is separately managed by the [RAIL_GetTxPower\(\)](#) / [RAIL_SetTxPower\(\)](#) APIs. Use this API to determine which values were set as a result of [RAIL_ConfigTxPower\(\)](#).

Definition at line 2858 of file `common/rail.h`

RAIL_SetTxPower

RAIL_Status_t RAIL_SetTxPower ([RAIL_Handle_t](#) railHandle, [RAIL_TxPowerLevel_t](#) powerLevel)

Set the TX power in units of raw units (see [rail_chip_specific.h](#) for value ranges).

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	powerLevel	Power in radio-specific RAIL_TxPowerLevel_t units.

Returns

- Status code indicating success of the function call.

To convert between decibels and the integer values that the registers take, call [RAIL_ConvertDbmToRaw\(\)](#). A weak version of this function, which works well with our boards is provided. However, customers using a custom board need to characterize radio operation on that board and override the function to convert appropriately from the desired dB values to raw integer values.

Depending on the configuration used in [RAIL_ConfigTxPower\(\)](#), not all power levels are achievable. This API will get as close as possible to the desired power without exceeding it, and calling [RAIL_GetTxPower\(\)](#) is the only way to know the exact value written.

Calling this function before configuring the PA (i.e., before a successful call to [RAIL_ConfigTxPower\(\)](#)) will return an error.

Definition at line 2886 of file `common/rail.h`

RAIL_GetTxPower

RAIL_TxPowerLevel_t RAIL_GetTxPower ([RAIL_Handle_t](#) railHandle)

Return the current power setting of the PA.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The radio-specific [RAIL_TxPowerLevel_t](#) value of the current transmit power.

This API returns the raw value that was set by [RAIL_SetTxPower\(\)](#). A weak version of [RAIL_ConvertRawToDbm\(\)](#) that works with Silicon Labs boards to convert the raw values into actual output dBm values is provided. However, customers using a custom board need to re-characterize the relationship between raw and decibel values and rewrite the provided function.

Calling this function before configuring the PA (i.e., before a successful call to [RAIL_ConfigTxPower\(\)](#)) will return error [RAIL_TX_POWER_LEVEL_INVALID](#).

Definition at line 2907 of file `common/rail.h`

RAIL_ConvertRawToDbm

`RAIL_TxPower_t` RAIL_ConvertRawToDbm ([RAIL_Handle_t](#) railHandle, [RAIL_TxPowerMode_t](#) mode, [RAIL_TxPowerLevel_t](#) powerLevel)

Convert raw values written to registers to decibel value (in units of deci-dBm).

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mode	PA mode for which to convert.
[in]	powerLevel	A raw amplifier register value to be converted to deci-dBm.

Returns

- raw amplifier values converted to units of deci-dBm.

A weak version of this function is provided that is tuned to provide accurate values for our boards. For a custom board, the relationship between what is written to the TX amplifier and the actual output power should be re-characterized and implemented in an overriding version of [RAIL_ConvertRawToDbm\(\)](#). For minimum code size and best speed, use only raw values with the [RAIL_SetTxPower\(\)](#) API and override this function with a smaller function. In the weak version provided with the RAIL library, railHandle is only used to indicate to the user from where the function was called, so it is okay to use either a real protocol handle, or one of the radio-generic ones, such as [RAIL_EFR32_HANDLE](#).

Although the implementation of this function may change, the signature must be as declared here.

Definition at line 2935 of file `common/rail.h`

RAIL_ConvertDbmToRaw

`RAIL_TxPowerLevel_t` RAIL_ConvertDbmToRaw ([RAIL_Handle_t](#) railHandle, [RAIL_TxPowerMode_t](#) mode, [RAIL_TxPower_t](#) power)

Convert the desired decibel value (in units of deci-dBm) to raw integer values used by the TX amplifier registers.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mode	PA mode for which to do the conversion.
[in]	power	Desired dBm value in units of deci-dBm.

Returns

- deci-dBm value converted to a raw integer value that can be used directly with [RAIL_SetTxPower\(\)](#).

A weak version of this function is provided that is tuned to provide accurate values for our boards. For a custom board, the relationship between what is written to the TX amplifier and the actual output power should be characterized and implemented in an overriding version of [RAIL_ConvertDbmToRaw\(\)](#). For minimum code size and best speed use only raw values with the [RAIL_SetTxPower\(\)](#) API and override this function with a smaller function. In the weak version provided with the RAIL library, railHandle is only used to indicate to the user from where the function was called, so it is okay to use either a real protocol handle, or one of the radio-generic ones, such as [RAIL_EFR32_HANDLE](#).

Although the implementation of this function may change, the signature must be as declared here.

Definition at line 2963 of file common/rail.h

RAIL_ConvertDbmToPowerSettingEntry

```
RAIL_Status_t RAIL_ConvertDbmToPowerSettingEntry (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPower_t power, RAIL_TxPowerSettingEntry_t * pPowerSettingInfo)
```

Converts the desired decibel value (in units of deci-dBm) to a RAIL_TxPowerSettingEntry_t.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mode	PA mode for which the conversion is to be done.
[in]	power	Desired dBm value in units of deci-dBm.
[out]	pPowerSettingInfo	A non-NULL pointer to the RAIL_TxPowerSettingEntry_t structure to be filled in with the converted value.

Returns

- Status code indicating success of function call.

A weak version of this function is provided that is tuned to provide accurate values for our boards. For a custom board, the relationship between what is written to the TX amplifier and the actual output power should be characterized and implemented in an overriding version of RAIL_ConvertDbmToPowerSettingEntry(). In the weak version provided with the RAIL library, railHandle is only used to indicate to the user from where the function was called, so it is okay to use either a real protocol handle, or one of the radio-generic ones, such as RAIL_EFR32_HANDLE.

Although the implementation of this function may change, the signature must be as declared here.

Definition at line 2992 of file common/rail.h

RAIL_VerifyTxPowerCurves

```
RAIL_Status_t RAIL_VerifyTxPowerCurves (const struct RAIL_TxPowerCurvesConfigAlt * config)
```

Verify the TX Power Curves on modules.

Parameters

[in]	config	A pointer to TX Power Curves to use on this module.

Returns

- Status code indicating success of function call.

This function only needs to be called when using a module and has no effect otherwise. Transmit will not work before this function is called.

Definition at line 3005 of file common/rail.h

RAIL_SetTxPowerDbm

```
RAIL_Status_t RAIL_SetTxPowerDbm (RAIL_Handle_t railHandle, RAIL_TxPower_t power)
```


Set the TX power in terms of deci-dBm instead of raw power level.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	power	A desired deci-dBm power to be set.

Returns

- Status code indicating success of the function call.

This is a utility function for user convenience. Normally, to set TX power in dBm, do the following:

```
RAIL_TxPower_t power = 100; // 100 deci-dBm, 10 dBm
RAIL_TxPowerConfig_t txPowerConfig;
RAIL_GetTxPowerConfig(railHandle, &txPowerConfig);
// RAIL_ConvertDbmToRaw() will be the weak version provided by Silicon Labs
// by default, or the customer version, if overwritten.
RAIL_TxPowerLevel_t powerLevel
    = RAIL_ConvertDbmToRaw(railHandle, txPowerConfig.mode, power);
RAIL_SetTxPower(railHandle, powerLevel);
```

This function wraps all those calls in a single function with power passed in as a parameter.

Definition at line 3029 of file common/rail.h

RAIL_GetTxPowerDbm

```
RAIL_TxPower_t RAIL_GetTxPowerDbm (RAIL_Handle_t railHandle)
```

Get the TX power in terms of deci-dBm instead of raw power level.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The current output power in deci-dBm.

This is a utility function for user convenience. Normally, to get TX power in dBm, do the following:

```
RAIL_TxPowerLevel_t powerLevel = RAIL_GetTxPower(railHandle);
RAIL_TxPowerConfig_t txPowerConfig;
RAIL_GetTxPowerConfig(railHandle, &txPowerConfig);
// RAIL_ConvertRawToDbm() will be the weak version provided by Silicon Labs
// by default, or the customer version, if overwritten.
RAIL_TxPower_t power
    = RAIL_ConvertRawToDbm(railHandle, txPowerConfig.mode, powerLevel);
return power;
```

This function wraps all those calls in a single function with power returned as the result.

Definition at line 3053 of file common/rail.h

RAIL_GetPowerSettingTable

```
const RAIL_PaPowerSetting_t * RAIL_GetPowerSettingTable (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode,
    RAIL_TxPower_t * minPower, RAIL_TxPower_t * maxPower, RAIL_TxPowerLevel_t * step)
```

Get the TX PA power setting table and related values.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mode	PA mode for which to get the powersetting table.
[out]	minPower	A pointer to a RAIL_TxPower_t .
[out]	maxPower	A pointer to a RAIL_TxPower_t .
[out]	step	A pointer to a RAIL_TxPowerLevel_t , but in deci-dBm units.

Returns

- Power setting table start address. When NULL is returned all out params above won't be set.

The number of entries in the table can be calculated based on output minPower, maxPower, and step parameters. For example, for minPower = 115 (11.5 dBm), maxPower = 300 (30 dBm), and step = 1 (0.1 dBm), the number of entries in table would be 186.

Definition at line 3071 of file common/rail.h

RAIL_SetPaPowerSetting

```
RAIL_Status_t RAIL_SetPaPowerSetting (RAIL_Handle_t railHandle, RAIL_PaPowerSetting_t paPowerSetting,
RAIL_TxPower_t minPowerDbm, RAIL_TxPower_t maxPowerDbm, RAIL_TxPower_t currentPowerDbm)
```

Set the TX PA power setting used to configure the PA hardware for the PA output power determined by [RAIL_SetTxPowerDbm\(\)](#).

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	paPowerSetting	The desired PA power setting.
[in]	minPowerDbm	The minimum power in dBm that the PA can output.
[in]	maxPowerDbm	The maximum power in dBm that the PA can output.
[in]	currentPowerDbm	The corresponding output power in dBm for this power setting.

Returns

- Status code indicating success of the function call.

Definition at line 3086 of file common/rail.h

RAIL_GetPaPowerSetting

```
RAIL_PaPowerSetting_t RAIL_GetPaPowerSetting (RAIL_Handle_t railHandle)
```

Get the TX PA power setting, which is used to configure power configurations when the dBm to paPowerSetting mapping table mode is used.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

The current PA power setting.

Definition at line 3099 of file common/rail.h

RAIL_EnablePaAutoMode

RAIL_Status_t RAIL_EnablePaAutoMode (RAIL_Handle_t railHandle, bool enable)

Enable automatic switching between PAs internally to the RAIL library.

Parameters

[in]	railHandle	A real RAIL instance handle.
[in]	enable	Enable or disable PA Auto Mode.

Returns

- Status code indicating success of the function call.

While PA Automode is enabled, the PA will be chosen and set automatically whenever [RAIL_SetTxPowerDbm\(\)](#) is called or whenever powers are coerced automatically, internally to the RAIL library during a channel change. While PA Auto Mode is enabled, users cannot call [RAIL_ConfigTxPower\(\)](#) or [RAIL_SetTxPower\(\)](#). When entering auto mode, [RAIL_SetTxPowerDbm\(\)](#) must be called to specify the desired power. When leaving auto mode, [RAIL_ConfigTxPower\(\)](#) as well as one of [RAIL_SetTxPower\(\)](#) or [RAIL_SetTxPowerDbm\(\)](#) must be called to re-specify the desired PA and power level combination.

Note

- : Power conversion curves must be initialized before calling this function. That is, [RAIL_ConvertDbmToRaw\(\)](#) and [RAIL_ConvertRawToDbm\(\)](#) must both be able to operate properly to ensure that PA Auto Mode functions correctly. See the PA Conversions plugin or AN1127 for more details.

Definition at line 3123 of file common/rail.h

RAIL_IsPaAutoModeEnabled

bool RAIL_IsPaAutoModeEnabled (RAIL_Handle_t railHandle)

Query status of PA Auto Mode.

Parameters

[in]	railHandle	A real RAIL instance handle.

Returns

- Indicator of whether Auto Mode is enabled (true) or not (false).

Definition at line 3131 of file common/rail.h

RAILCb_PaAutoModeDecision

RAIL_Status_t RAILCb_PaAutoModeDecision (RAIL_Handle_t railHandle, RAIL_TxPower_t * power, RAIL_TxPowerMode_t * mode, const RAIL_ChannelConfigEntry_t * chCfgEntry)

Callback that decides which PA and power level should be used while in PA auto mode.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	power	A pointer to the dBm output power (in deci-dBm, 10*dBm) being requested. The value this points to when the function returns will be applied to the radio.
[out]	mode	A pointer to the RAIL_TxPowerMode_t to be used to achieve the requested power. The value this points to when the function returns will be applied to the radio.
[in]	chCfgEntry	A pointer to a RAIL_ChannelConfigEntry_t . While switching channels, it will be the entry RAIL is switch to , during a call to RAIL_SetTxPowerDbm() , it will be the entry RAIL is already on . Can be NULL if a channel configuration was not set or no valid channels are present.

Returns

- Status code indicating success of the function call. If this is anything except [RAIL_STATUS_NO_ERROR](#), neither PA's nor their powers will be configured automatically.

Whatever values mode and powerLevel point to when this function return will be applied to the PA hardware and used for transmits.

Note

- The mode and power level provided by this function depends on the [RAIL_PaAutoModeConfig](#) provided for the radio. The [RAIL_PaAutoModeConfig](#) definition for a radio should tend to cover all the bands supported by the radio and cover the full range of power in each to find a valid entry for requested power for a specific band.

Definition at line 3162 of file `common/rail.h`

RAIL_ConfigPaAutoEntry

```
RAIL_Status_t RAIL_ConfigPaAutoEntry (RAIL\_Handle\_t railHandle, const RAIL\_PaAutoModeConfigEntry\_t *
paAutoModeEntry)
```

Configure the PA auto mode entries.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	paAutoModeEntry	A pointer to entries used to configure PA auto mode decision points. The final entry must set its RAIL_PaAutoModeConfigEntry_t::band to RAIL_PA_BAND_COUNT .

Returns

- Status parameter indicating success of function call.

Definition at line 97 of file `plugin/pa-auto-mode/pa_auto_mode.h`

sl_rail_util_pa_nvm_write_mode

```
sl_status_t sl_rail_util_pa_nvm_write_mode (RAIL\_TxPowerMode\_t pa_mode)
```

Store a [RAIL_TxPowerMode_t](#) into NVM.

Parameters

[in]	pa_mode	The PA mode.

Returns

Status code indicating success of the function call.

Definition at line 173 of file plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h

sl_rail_util_pa_nvm_read_mode

RAIL_TxPowerMode_t sl_rail_util_pa_nvm_read_mode (void)

Reads stored RAIL_TxPowerMode_t from NVM.

Parameters

N/A		

Returns

- The stored PA mode, or RAIL_TX_POWER_MODE_NONE if no stored PA mode exists or is invalid.

Definition at line 181 of file plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h

sl_rail_util_pa_nvm_write_config

sl_status_t sl_rail_util_pa_nvm_write_config (const sl_rail_nvm_pa_config_t * p_pa_config)

Stores a sl_rail_nvm_pa_config_t into NVM.

Parameters

[in]	p_pa_config	A non-NULL pointer to PA curves/table config.

Returns

- Status code indicating success of the function call.

Definition at line 188 of file plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h

sl_rail_util_pa_nvm_read_config

sl_status_t sl_rail_util_pa_nvm_read_config (sl_rail_nvm_pa_config_t * p_pa_config)

Reads stored sl_rail_nvm_pa_config_t from NVM.

Parameters

[out]	p_pa_config	A non-NULL pointer to the PA curves/table config read from NVM.

Returns

- Status code indicating success of the function call.

Definition at line 196 of file plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h

RAIL_TxPowerSettingEntry_t

A structure containing power-setting information for a deci-dBm power.

Public Attributes

RAIL_PaPowerSetting_t	paPowerSetting PowerSetting information corresponds to currentPaPowerDdbm.
RAIL_TxPower_t	minPaPowerDdbm Minimum power (in deci-dBm) supported by powersetting table in use.
RAIL_TxPower_t	maxPaPowerDdbm Maximum power (in deci-dBm) supported by powersetting table in use.
RAIL_TxPower_t	currentPaPowerDdbm Current power (in deci-dBm)

Public Attribute Documentation

paPowerSetting

RAIL_PaPowerSetting_t RAIL_TxPowerSettingEntry_t::paPowerSetting

PowerSetting information corresponds to currentPaPowerDdbm.

Definition at line 1784 of file common/rail_types.h

minPaPowerDdbm

RAIL_TxPower_t RAIL_TxPowerSettingEntry_t::minPaPowerDdbm

Minimum power (in deci-dBm) supported by powersetting table in use.

Definition at line 1786 of file common/rail_types.h

maxPaPowerDdbm

RAIL_TxPower_t RAIL_TxPowerSettingEntry_t::maxPaPowerDdbm

Maximum power (in deci-dBm) supported by powersetting table in use.

Definition at line 1788 of file common/rail_types.h

currentPaPowerDdbm

RAIL_TxPower_t RAIL_TxPowerSettingEntry_t::currentPaPowerDdbm

Current power (in deci-dBm)

Definition at line 1790 of file common/rail_types.h

RAIL_TxPowerConfig_t

A structure containing values used to initialize the power amplifiers.

Public Attributes

RAIL_TxPowerMode_t	mode	TX power mode.
uint16_t	voltage	Power amplifier supply voltage in mV, generally: DCDC supply ~ 1800 mV (1.8 V) Battery supply ~ 3300 mV (3.3 V)
uint16_t	rampTime	The amount of time to spend ramping for TX in microseconds.

Public Attribute Documentation

mode

RAIL_TxPowerMode_t RAIL_TxPowerConfig_t::mode

TX power mode.

Definition at line 1938 of file common/rail_types.h

voltage

uint16_t RAIL_TxPowerConfig_t::voltage

Power amplifier supply voltage in mV, generally: DCDC supply ~ 1800 mV (1.8 V) Battery supply ~ 3300 mV (3.3 V)

Definition at line 1944 of file common/rail_types.h

rampTime

uint16_t RAIL_TxPowerConfig_t::rampTime

The amount of time to spend ramping for TX in microseconds.

Definition at line 1946 of file common/rail_types.h

RAIL_PaAutoModeConfigEntry_t

Struct to ease specification of appropriate ranges within which a PA should be used.

Public Attributes

RAIL_TxPower_t	min	The minimum (inclusive) deci-dBm power to use with this entry.
RAIL_TxPower_t	max	The maximum (inclusive) deci-dBm power to use with this entry.
RAIL_TxPowerMode_t	mode	The PA that this range of powers applies to.
RAIL_PaBand_t	band	The RF band that this PA works with.

Public Attribute Documentation

min

RAIL_TxPower_t RAIL_PaAutoModeConfigEntry_t::min

The minimum (inclusive) deci-dBm power to use with this entry.

Definition at line 74 of file plugin/pa-auto-mode/pa_auto_mode.h

max

RAIL_TxPower_t RAIL_PaAutoModeConfigEntry_t::max

The maximum (inclusive) deci-dBm power to use with this entry.

Definition at line 76 of file plugin/pa-auto-mode/pa_auto_mode.h

mode

RAIL_TxPowerMode_t RAIL_PaAutoModeConfigEntry_t::mode

The PA that this range of powers applies to.

Definition at line 78 of file plugin/pa-auto-mode/pa_auto_mode.h

band

RAIL_PaBand_t RAIL_PaAutoModeConfigEntry_t::band

The RF band that this PA works with.

Definition at line 80 of file plugin/pa-auto-mode/pa_auto_mode.h

sl_rail_nvm_pa_curve_t

NVM-suitable PA curve structure to map to a [RAIL_TxPowerCurveAlt_t](#).
The size of this structure must be a multiple of 4 bytes.

Public Attributes

int16_t	curve_min_ddbm	Minimum deci-dBm value.
int16_t	curve_max_ddbm	Maximum deci-dBm value.
RAIL_TxPowerCurveSegment_t	curve_segments	Curve segments.

Public Attribute Documentation

curve_min_ddbm

```
int16_t sl_rail_nvm_pa_curve_t::curve_min_ddbm
```

Minimum deci-dBm value.

Definition at line 92 of file `plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h`

curve_max_ddbm

```
int16_t sl_rail_nvm_pa_curve_t::curve_max_ddbm
```

Maximum deci-dBm value.

Definition at line 94 of file `plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h`

curve_segments

```
RAIL_TxPowerCurveSegment_t sl_rail_nvm_pa_curve_t::curve_segments[9]
```

Curve segments.

Definition at line 96 of file `plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h`

sl_rail_nvm_pa_table_t

NVM-suitable PA mapping table structure to map to a [RAIL_PowerConversion_t::mappingTable](#).

The size of this structure must be a multiple of 4 bytes.

Public Attributes

int16_t

[ddbm_values](#)

Mapping table of deci-dBm power values.

Public Attribute Documentation

ddbm_values

int16_t sl_rail_nvm_pa_table_t::ddbm_values[16]

Mapping table of deci-dBm power values.

Definition at line 108 of file `plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h`

sl_rail_nvm_pa_descriptor_t

NVM-suitable PA descriptor structure to map to a [RAIL_PaDescriptor_t](#).

The size of this structure must be a multiple of 4 bytes.

Public Attributes

RAIL_PaConversionAlgorithm_t	algorithm Algorithm used to map dBm to power levels for this PA.
uint8_t	num_segments_or_entries The number of curve piecewise segments or mapping table entries provided for the PA according to its algorithm.
RAIL_TxPowerLevel_t	min Minimum power level for this PA.
RAIL_TxPowerLevel_t	max Maximum power level for this PA.

Public Attribute Documentation

algorithm

`RAIL_PaConversionAlgorithm_t sl_rail_nvm_pa_descriptor_t::algorithm`

Algorithm used to map dBm to power levels for this PA.

Definition at line 120 of file `plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h`

num_segments_or_entries

`uint8_t sl_rail_nvm_pa_descriptor_t::num_segments_or_entries`

The number of curve piecewise segments or mapping table entries provided for the PA according to its algorithm.

This should not exceed [SL_RAIL_NVM_PA_CURVE_SEGMENTS](#) for a PA using [RAIL_PA_ALGORITHM_PIECEWISE_LINEAR](#) or [SL_RAIL_NVM_PA_TABLE_ENTRIES](#) for a PA using [RAIL_PA_ALGORITHM_MAPPING_TABLE](#).

Definition at line 129 of file `plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h`

min

`RAIL_TxPowerLevel_t sl_rail_nvm_pa_descriptor_t::min`

Minimum power level for this PA.

Definition at line 131 of file `plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h`

max

RAIL_TxPowerLevel_t sl_rail_nvm_pa_descriptor_t::max

Maximum power level for this PA.

Definition at line 133 of file plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h

sl_rail_nvm_pa_config_t

NVM-suitable PA curves/tables configuration structure to map to a [RAIL_TxPowerCurvesConfigAlt_t](#).

Public Attributes

uint8_t	version	Version number of this structure.
uint8_t	num_descriptors	Number of descriptors described below in pa_descriptors[].
uint16_t	pa_voltage	PA VDD voltage, in millivolts.
uint32_t	signature	Signature used for validation of the curves configuration.
sl_rail_nvm_pa_descriptor_t	pa_descriptors	The PA data descriptor, one for each PA.
sl_rail_nvm_pa_curve_t	curve	PA piecewise-linear curve.
sl_rail_nvm_pa_table_t	table	PA raw to dBm mapping table.
union sl_rail_nvm_pa_config_t::@1	pa_curve_or_table	Union for either a curve or table.

Public Attribute Documentation

version

```
uint8_t sl_rail_nvm_pa_config_t::version
```

Version number of this structure.

Should correspond to [SL_RAIL_NVM_PA_CONFIGS_VERSION](#).

Definition at line 146 of file `plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h`

num_descriptors

```
uint8_t sl_rail_nvm_pa_config_t::num_descriptors
```

Number of descriptors described below in pa_descriptors[].

Must equal [SL_RAIL_NVM_PA_COUNT](#).

Definition at line 151 of file `plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h`

pa_voltage

```
uint16_t sl_rail_nvm_pa_config_t::pa_voltage
```

PA VDD voltage, in millivolts.

Definition at line 153 of file `plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h`

signature

```
uint32_t sl_rail_nvm_pa_config_t::signature
```

Signature used for validation of the curves configuration.

Definition at line 155 of file `plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h`

pa_descriptors

```
sl_rail_nvm_pa_descriptor_t sl_rail_nvm_pa_config_t::pa_descriptors[RAIL_NUM_PA]
```

The PA data descriptor, one for each PA.

Definition at line 157 of file `plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h`

curve

```
sl_rail_nvm_pa_curve_t sl_rail_nvm_pa_config_t::curve
```

PA piecewise-linear curve.

Definition at line 161 of file `plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h`

table

```
sl_rail_nvm_pa_table_t sl_rail_nvm_pa_config_t::table
```

PA raw to dBm mapping table.

Definition at line 163 of file `plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h`

pa_curve_or_table

```
union sl_rail_nvm_pa_config_t::@1 sl_rail_nvm_pa_config_t::pa_curve_or_table[ RAIL_NUM_PA ]
```

Union for either a curve or table.

Definition at line 164 of file `plugin/pa-conversions/nvm/sl_rail_util_pa_nvm_configs.h`

EFR32xG2x

EFR32xG2x

Types specific to the EFR32xG2x for dealing with the on-chip PAs.

Macros

#define	RAIL_TX_POWER_LEVEL_2P4_HP_MAX (240)	The maximum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4GIG_HP mode.
#define	RAIL_TX_POWER_LEVEL_2P4_HP_MIN (1U)	The minimum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4GIG_HP mode.
#define	RAIL_TX_POWER_LEVEL_2P4_MP_MAX (RAIL_TX_POWER_LEVEL_2P4_HP_MAX)	The maximum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4GIG_MP mode.
#define	RAIL_TX_POWER_LEVEL_2P4_MP_MIN (RAIL_TX_POWER_LEVEL_2P4_HP_MIN)	The maximum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4GIG_MP mode.
#define	RAIL_TX_POWER_LEVEL_2P4_LP_MAX (RAIL_TX_POWER_LEVEL_2P4_HP_MAX)	The maximum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4GIG_LP mode.
#define	RAIL_TX_POWER_LEVEL_2P4_LP_MIN (RAIL_TX_POWER_LEVEL_2P4_HP_MIN)	The minimum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4GIG_LP mode.
#define	RAIL_TX_POWER_LEVEL_2P4_LL_P_MAX (RAIL_TX_POWER_LEVEL_2P4_HP_MAX)	The maximum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4GIG_LL_P mode.
#define	RAIL_TX_POWER_LEVEL_2P4_LL_P_MIN (RAIL_TX_POWER_LEVEL_2P4_HP_MIN)	The minimum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4GIG_LL_P mode.
#define	RAIL_SUBGIG_MAX 0U	The maximum valid value for the RAIL_TxPowerLevel_t when using a Sub-GHz PA mode.
#define	RAIL_SUBGIG_MIN 1U	The minimum valid value for the RAIL_TxPowerLevel_t when using a Sub-GHz PA mode.
#define	RAIL_TX_POWER_LEVEL_SUBGIG_HP_MAX (RAIL_SUBGIG_MAX)	The maximum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_SUBGIG_HP mode.
#define	RAIL_TX_POWER_LEVEL_SUBGIG_HP_MIN (RAIL_SUBGIG_MIN)	The minimum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_SUBGIG_HP mode.
#define	RAIL_TX_POWER_LEVEL_SUBGIG_MP_MAX (RAIL_SUBGIG_MAX)	The maximum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_SUBGIG_MP mode.
#define	RAIL_TX_POWER_LEVEL_SUBGIG_MP_MIN (RAIL_SUBGIG_MIN)	The minimum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_SUBGIG_MP mode.
#define	RAIL_TX_POWER_LEVEL_SUBGIG_LP_MAX (RAIL_SUBGIG_MAX)	The maximum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_SUBGIG_LP mode.
#define	RAIL_TX_POWER_LEVEL_SUBGIG_LP_MIN (RAIL_SUBGIG_MIN)	The minimum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_SUBGIG_LP mode.

#define RAIL_TX_POWER_LEVEL_SUBGIG_LL_P_MAX (RAIL_SUBGIG_MAX)
The maximum valid value for the [RAIL_TxPowerLevel_t](#) when in [RAIL_TX_POWER_MODE_SUBGIG_LL_P](#) mode.

#define RAIL_TX_POWER_LEVEL_SUBGIG_LL_P_MIN (RAIL_SUBGIG_MIN)
The minimum valid value for the [RAIL_TxPowerLevel_t](#) when in [RAIL_TX_POWER_MODE_SUBGIG_LL_P](#) mode.

#define RAIL_TX_POWER_LEVEL_HP_MAX RAIL_TX_POWER_LEVEL_2P4_HP_MAX
Backwards compatability define.

#define RAIL_TX_POWER_LEVEL_HP_MIN RAIL_TX_POWER_LEVEL_2P4_HP_MIN
Backwards compatability define.

#define RAIL_TX_POWER_LEVEL_MP_MAX RAIL_TX_POWER_LEVEL_2P4_MP_MAX
Backwards compatability define.

#define RAIL_TX_POWER_LEVEL_MP_MIN RAIL_TX_POWER_LEVEL_2P4_MP_MIN
Backwards compatability define.

#define RAIL_TX_POWER_LEVEL_LP_MAX RAIL_TX_POWER_LEVEL_2P4_LP_MAX
Backwards compatability define.

#define RAIL_TX_POWER_LEVEL_LP_MIN RAIL_TX_POWER_LEVEL_2P4_LP_MIN
Backwards compatability define.

#define RAIL_TX_POWER_LEVEL_SUBGIG_MAX RAIL_TX_POWER_LEVEL_SUBGIG_HP_MAX
Backwards compatability define.

#define RAIL_TX_POWER_LEVEL_SUBGIG_MIN RAIL_TX_POWER_LEVEL_SUBGIG_HP_MIN
Backwards compatability define.

#define RAIL_NUM_PA (3U)
The number of PA's on this chip (including Virtual PAs).

#define RAIL_POWER_MODE_IS_ANY_DBM_POWERSETTING_MAPPING_TABLE (x)
Convenience macro for any mapping table mode.

#define RAIL_POWER_MODE_SUPPORTS_RAW_SETTING (x)
Convenience macro to check if the power mode supports raw setting.

PA Curve Conversions

Modules

RAIL_TxPowerCurveSegment_t	
RAIL_TxPowerCurves_t	
RAIL_TxPowerCurvesConfig_t	
RAIL_TxPowerCurveAlt_t	
RAIL_PowerConversion_t	
RAIL_PaDescriptor_t	
RAIL_TxPowerCurvesConfigAlt_t	

Enumerations

enum	RAIL_PaConversionAlgorithm_t { RAIL_PA_ALGORITHM_PIECEWISE_LINEAR RAIL_PA_ALGORITHM_MAPPING_TABLE RAIL_PA_ALGORITHM_DBM_POWERSETTING_MAPPING_TABLE RAIL_PA_ALGORITHM_COUNT }
	PA conversion algorithms types for converting between dBm and power levels.

Variables

const RAIL_TxPowerCurvesConfigAlt_t	RAIL_TxPowerCurvesVbat The curves to be used when battery voltage powers transmission.
const RAIL_TxPowerCurvesConfigAlt_t	RAIL_TxPowerCurvesDcdc The curves to be used when the DC-DC converter powers transmission.

Functions

RAIL_Status_t	RAIL_InitTxPowerCurves (const RAIL_TxPowerCurvesConfig_t *config) Initialize Transmit power curves.
RAIL_Status_t	RAIL_InitTxPowerCurvesAlt (const RAIL_TxPowerCurvesConfigAlt_t *config) Initialize TxPower curves.
RAIL_TxPowerCurves_t const *	RAIL_GetTxPowerCurve (RAIL_TxPowerMode_t mode) Gets the curve that should be used for conversion functions based on the current PA configuration.
RAIL_Status_t	RAIL_GetTxPowerCurveLimits (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPower_t *maxpower, RAIL_TxPower_t *increment) Gets the maximum power in deci-dBm that should be used for calculating the segments and to find right curve segment to convert Dbm to raw power level for a specific PA.

void	<code>sl_rail_util_pa_init(void)</code> SL_RAIL_3_API.
<code>RAIL_TxPowerConfig_t *</code>	<code>sl_rail_util_pa_get_tx_power_config_2p4ghz(void)</code> Get a pointer to the TX Power Config 2.4 GHz structure.
<code>RAIL_TxPowerConfig_t *</code>	<code>sl_rail_util_pa_get_tx_power_config_subghz(void)</code> Get a pointer to the TX Power Config Sub-GHz structure.
<code>RAIL_TxPowerConfig_t *</code>	<code>sl_rail_util_pa_get_tx_power_config_ofdm(void)</code> Get a pointer to the TX Power Config OFDM structure.
void	<code>sl_rail_util_pa_on_channel_config_change(RAIL_Handle_t rail_handle, const RAIL_ChannelConfigEntry_t *entry)</code> Provide a channel config change callback capable of configuring the PA correctly.

Enumeration Documentation

RAIL_PaConversionAlgorithm_t

```
RAIL_PaConversionAlgorithm_t
```

PA conversion algorithms types for converting between dBm and power levels.

Enumerator	
RAIL_PA_ALGORITHM_PIECEWISE_LINEAR	Piecewise linear fit.
RAIL_PA_ALGORITHM_MAPPING_TABLE	Mapping table between quantities.
RAIL_PA_ALGORITHM_DBM_POWERSETTING_MAPPING_TABLE	Mapping table between pa power settings and dBm values.
RAIL_PA_ALGORITHM_COUNT	Algorithm count.

Definition at line 152 of file `plugin/pa-conversions/pa_curve_types_efr32.h`

Variable Documentation

RAIL_TxPowerCurvesVbat

```
const RAIL_TxPowerCurvesConfigAlt_t RAIL_TxPowerCurvesVbat
```

The curves to be used when battery voltage powers transmission.

Definition at line 87 of file `plugin/pa-conversions/pa_conversions_efr32.h`

RAIL_TxPowerCurvesDcdc

```
const RAIL_TxPowerCurvesConfigAlt_t RAIL_TxPowerCurvesDcdc
```

The curves to be used when the DC-DC converter powers transmission.

Definition at line 90 of file `plugin/pa-conversions/pa_conversions_efr32.h`

Function Documentation

RAIL_InitTxPowerCurves

```
RAIL_Status_t RAIL_InitTxPowerCurves (const RAIL\_TxPowerCurvesConfig\_t * config)
```

Initialize Transmit power curves.

Parameters

[in]	config	A pointer to the custom TX power curves.

Returns

- Status code indicating success of the function call.

Deprecatedfunction is no longer supported. Must use [RAIL_InitTxPowerCurvesAlt\(\)](#) instead.

Definition at line 101 of file `plugin/pa-conversions/pa_conversions_efr32.h`

RAIL_InitTxPowerCurvesAlt

```
RAIL_Status_t RAIL_InitTxPowerCurvesAlt (const RAIL\_TxPowerCurvesConfigAlt\_t * config)
```

Initialize TxPower curves.

Parameters

[in]	config	A pointer to the custom TX power curves to use.

Returns

- Status code indicating success of the function call.

Definition at line 109 of file `plugin/pa-conversions/pa_conversions_efr32.h`

RAIL_GetTxPowerCurve

```
RAIL_TxPowerCurves_t const * RAIL_GetTxPowerCurve (RAIL\_TxPowerMode\_t mode)
```

Gets the curve that should be used for conversion functions based on the current PA configuration.

Parameters

[in]	mode	PA mode whose curves are needed.

Returns

- A pointer to the [RAIL_TxPowerCurves_t](#) that are used for conversion functions.

Note

- : If the mode is not supported by the the chip, then NULL will be returned.

Definition at line 121 of file `plugin/pa-conversions/pa_conversions_efr32.h`

RAIL_GetTxPowerCurveLimits

```
RAIL_Status_t RAIL_GetTxPowerCurveLimits (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPower_t * maxpower, RAIL_TxPower_t * increment)
```

Gets the maximum power in deci-dBm that should be used for calculating the segments and to find right curve segment to convert Dbm to raw power level for a specific PA.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mode	PA mode whose curves are needed.
[out]	maxpower	A non-NULL pointer to memory allocated to hold the max power in deci-dBm used in calculation of curve segments.
[out]	increment	A non-NULL pointer to memory allocated to hold the increment in deci-dBm used in calculation of curve segments.

Returns

- Status code indicating success of the function call.

For the PAs with [RAIL_PaConversionAlgorithm_tRAIL_PA_ALGORITHM_PIECEWISE_LINEAR](#), if the curves are generated with maxPower and increment other than [RAIL_TX_POWER_CURVE_DEFAULT_MAX](#) and [RAIL_TX_POWER_CURVE_DEFAULT_INCREMENT](#) respectively, then the first [RAIL_TxPowerCurveSegment_t](#) has its maxPowerLevel equal to [RAIL_TX_POWER_LEVEL_INVALID](#) and its slope and intercept stores the maxPower and increment in deci-dBm respectively.

Definition at line 144 of file `plugin/pa-conversions/pa_conversions_efr32.h`

sl_rail_util_pa_init

```
void sl_rail_util_pa_init (void )
```

SL_RAIL_3_API.

Parameters

N/A		

Initialize PA TX Curves.

Definition at line 156 of file `plugin/pa-conversions/pa_conversions_efr32.h`

sl_rail_util_pa_get_tx_power_config_2p4ghz

```
RAIL_TxPowerConfig_t * sl_rail_util_pa_get_tx_power_config_2p4ghz (void )
```

Get a pointer to the TX Power Config 2.4 GHz structure.

Parameters

N/A		

Returns

- A pointer to the TX Power Config stucture.

Definition at line 163 of file `plugin/pa-conversions/pa_conversions_efr32.h`

sl_rail_util_pa_get_tx_power_config_subghz

```
RAIL_TxPowerConfig_t * sl_rail_util_pa_get_tx_power_config_subghz (void )
```

Get a pointer to the TX Power Config Sub-GHz structure.

Parameters

N/A		

Returns

- A pointer to the TX Power Config stucture.

Definition at line 170 of file `plugin/pa-conversions/pa_conversions_efr32.h`

sl_rail_util_pa_get_tx_power_config_ofdm

```
RAIL_TxPowerConfig_t * sl_rail_util_pa_get_tx_power_config_ofdm (void )
```

Get a pointer to the TX Power Config OFDM structure.

Parameters

N/A		

Returns

- A pointer to the TX Power Config stucture.

Definition at line 177 of file `plugin/pa-conversions/pa_conversions_efr32.h`

sl_rail_util_pa_on_channel_config_change

```
void sl_rail_util_pa_on_channel_config_change (RAIL_Handle_t rail_handle, const RAIL_ChannelConfigEntry_t * entry)
```

Provide a channel config change callback capable of configuring the PA correctly.

Parameters

[in]	rail_handle	The RAIL handle being passed into this callback.
[in]	entry	A pointer to the channel config entry being switched to by hardware.

Definition at line 187 of file `plugin/pa-conversions/pa_conversions_efr32.h`

RAIL_TxPowerCurveSegment_t

Structure containing data defining each segment of the deci-dBm to raw power level mapping curve fits.

Note, these used in an equation of the form:

$$\text{powerLevel} * 1000 = \text{slope} * \text{power} + \text{intercept}$$

powerLevel is the 0-252/0-248/1-7 values used in the RAIL_Get/SetTxPower() functions, and power is the actual output power of the PA, specified in deci-dBm.

Note

- If the curves are generated with maxPower and increment other than [RAIL_TX_POWER_CURVE_DEFAULT_MAX](#) and [RAIL_TX_POWER_CURVE_DEFAULT_INCREMENT](#) respectively, then the first [RAIL_TxPowerCurveSegment_t](#) has its maxPowerLevel equal to [RAIL_TX_POWER_LEVEL_INVALID](#) and its slope and intercept stores the maxPower and increment in deci-dBm respectively.

Public Attributes

uint16_t	maxPowerLevel	The highest power level that this segment will be used to convert.
int16_t	slope	slope of the line
int32_t	intercept	y-intercept of the line

Public Attribute Documentation

maxPowerLevel

uint16_t RAIL_TxPowerCurveSegment_t::maxPowerLevel

The highest power level that this segment will be used to convert.

Definition at line 76 of file `plugin/pa-conversions/pa_curve_types_efr32.h`

slope

int16_t RAIL_TxPowerCurveSegment_t::slope

slope of the line

Definition at line 78 of file `plugin/pa-conversions/pa_curve_types_efr32.h`

intercept

int32_t RAIL_TxPowerCurveSegment_t::intercept

y-intercept of the line

Definition at line 80 of file plugin/pa-conversions/pa_curve_types_efr32.h

RAIL_TxPowerCurves_t

Structure containing the min and max values for a given PA and voltage supply combination (in deci-dBm).

Public Attributes

int16_t	maxPower	max deci-dBm value
int16_t	minPower	min deci-dBm value
const RAIL_TxPowerCurveSegment_t *	powerParams	Pointer to an array of RAIL_TxPowerCurvesConfig_t::piecewiseSegments elements of RAIL_TxPowerCurveSegment_t for deci-dBm to raw power level conversion fits.

Public Attribute Documentation

maxPower

```
int16_t RAIL_TxPowerCurves_t::maxPower
```

max deci-dBm value

Definition at line 91 of file `plugin/pa-conversions/pa_curve_types_efr32.h`

minPower

```
int16_t RAIL_TxPowerCurves_t::minPower
```

min deci-dBm value

Definition at line 93 of file `plugin/pa-conversions/pa_curve_types_efr32.h`

powerParams

```
const RAIL_TxPowerCurveSegment_t* RAIL_TxPowerCurves_t::powerParams
```

Pointer to an array of [RAIL_TxPowerCurvesConfig_t::piecewiseSegments](#) elements of [RAIL_TxPowerCurveSegment_t](#) for deci-dBm to raw power level conversion fits.

Definition at line 99 of file `plugin/pa-conversions/pa_curve_types_efr32.h`

RAIL_TxPowerCurvesConfig_t

Structure containing curve fit information and other metadata required to properly use the WEAK versions of RAIL_ConvertRawToDbm and RAIL_ConvertDbmToRaw.

Public Attributes

<code>const RAIL_TxPowerCurves_t *</code>	<code>txPower24HpCurves</code> Pointer a RAIL_TxPowerCurves_t representing the piecewise linear segments of curves that map power level to power in dBm for the 2.4 GHz high power PA.
<code>const RAIL_TxPowerCurves_t *</code>	<code>txPowerSgCurves</code> Pointer a RAIL_TxPowerCurves_t representing the piecewise linear segments of curves that map power level to power in dBm for the subgig PA.
<code>const int16_t *</code>	<code>txPower24LpCurves</code> Look up table for each of the power levels of the 2.4GHz low power amplifier and their equivalent deci-dB value.
<code>uint8_t</code>	<code>piecewiseSegments</code> The number of piecewise segments provided to the PA in each of the four conversion curve fits.

Public Attribute Documentation

txPower24HpCurves

```
const RAIL_TxPowerCurves_t* RAIL_TxPowerCurvesConfig_t::txPower24HpCurves
```

Pointer a [RAIL_TxPowerCurves_t](#) representing the piecewise linear segments of curves that map power level to power in dBm for the 2.4 GHz high power PA.

Note

- By the default conversion implementation, segments must be specified in decreasing power order. That is, the 0th entry of this array should be used to convert the highest power (levels). Segment at position n is valid from maxPowerLevel+1 from the segment at n+1 (or 0 if n is array length - 1) to maxPowerLevel of segment n, inclusive.

Definition at line 121 of file `plugin/pa-conversions/pa_curve_types_efr32.h`

txPowerSgCurves

```
const RAIL_TxPowerCurves_t* RAIL_TxPowerCurvesConfig_t::txPowerSgCurves
```

Pointer a [RAIL_TxPowerCurves_t](#) representing the piecewise linear segments of curves that map power level to power in dBm for the subgig PA.

Note

- By the default conversion implementation, segments must be specified in decreasing power order. That is, the 0th entry of this array should be used to convert the highest power (levels). Segment at position n is valid from maxPowerLevel+1 from the segment at n+1 (or 0 if n is array length - 1) to maxPowerLevel of segment n, inclusive.

Definition at line 133 of file `plugin/pa-conversions/pa_curve_types_efr32.h`

txPower24LpCurves

```
const int16_t* RAIL_TxPowerCurvesConfig_t::txPower24LpCurves
```

Look up table for each of the power levels of the 2.4GHz low power amplifier and their equivalent deci-dB value.

Definition at line 139 of file `plugin/pa-conversions/pa_curve_types_efr32.h`

piecewiseSegments

```
uint8_t RAIL_TxPowerCurvesConfig_t::piecewiseSegments
```

The number of piecewise segments provided to the PA in each of the four conversion curve fits.

The default is 8, but regardless of the number, it must be the same for all curves.

Definition at line 145 of file `plugin/pa-conversions/pa_curve_types_efr32.h`

RAIL_TxPowerCurveAlt_t

Structure containing the min and max values for a given PA and voltage supply combination (in deci-dBm).

Public Attributes

<code>int16_t</code>	<code>maxPower</code> max deci-dBm value
<code>int16_t</code>	<code>minPower</code> min deci-dBm value
<code>RAIL_TxPowerCurveSegment_t</code>	<code>powerParams</code> Array of <code>RAIL_PaDescriptor_t::segmentsRAIL_TxPowerCurveSegment_t</code> structures for the deci-dBm to raw power level conversion fits.

Public Attribute Documentation

maxPower

```
int16_t RAIL_TxPowerCurveAlt_t::maxPower
```

max deci-dBm value

Definition at line 179 of file plugin/pa-conversions/pa_curve_types_efr32.h

minPower

```
int16_t RAIL_TxPowerCurveAlt_t::minPower
```

min deci-dBm value

Definition at line 181 of file plugin/pa-conversions/pa_curve_types_efr32.h

powerParams

```
RAIL_TxPowerCurveSegment_t RAIL_TxPowerCurveAlt_t::powerParams[]
```

Array of `RAIL_PaDescriptor_t::segmentsRAIL_TxPowerCurveSegment_t` structures for the deci-dBm to raw power level conversion fits.

Definition at line 188 of file plugin/pa-conversions/pa_curve_types_efr32.h

RAIL_PowerConversion_t

Union containing a pointer to algorithm-specific conversion data.

Public Attributes

<code>const RAIL_TxPowerCurveAlt_t *</code>	<code>powerCurve</code> Pointer to a powerCurve containing line segment data for the curves corresponding to a specific PA.
<code>const int16_t *</code>	<code>mappingTable</code> Lookup table for PA's which use the mapping table algorithm for converting between deci-dBm and power levels.

Public Attribute Documentation

powerCurve

```
const RAIL_TxPowerCurveAlt_t* RAIL_PowerConversion_t::powerCurve
```

Pointer to a powerCurve containing line segment data for the curves corresponding to a specific PA.

Note

- By the default conversion implementation, segments must be specified in decreasing power order. That is, the 0th entry of this array should be used to convert the highest power (levels). Segment at position n is valid from maxPowerLevel+1 from the segment at n+1 (or 0 if n is array length - 1) to maxPowerLevel of segment n, inclusive.

Definition at line 225 of file plugin/pa-conversions/pa_curve_types_efr32.h

mappingTable

```
const int16_t* RAIL_PowerConversion_t::mappingTable
```

Lookup table for PA's which use the mapping table algorithm for converting between deci-dBm and power levels.

Definition at line 237 of file plugin/pa-conversions/pa_curve_types_efr32.h

RAIL_PaDescriptor_t

Struct containing specifics of PA configuration.

PA descriptor as used in the PA conversion functions.

Public Attributes

RAIL_PaConversionAlgorithm_t	algorithm Algorithm used to map dBm to power levels for this PA.
uint8_t	segments The number of piecewise segments provided to the PA in a piecewise linear curve fit.
RAIL_TxPowerLevel_t	min Min power level for this PA.
RAIL_TxPowerLevel_t	max Max power level for this PA.
RAIL_PowerConversion_t	conversion Union containing a pointer to algorithm-specific conversion data.

Public Attribute Documentation

algorithm

RAIL_PaConversionAlgorithm_t RAIL_PaDescriptor_t::algorithm

Algorithm used to map dBm to power levels for this PA.

Definition at line 249 of file plugin/pa-conversions/pa_curve_types_efr32.h

segments

uint8_t RAIL_PaDescriptor_t::segments

The number of piecewise segments provided to the PA in a piecewise linear curve fit.

The default is 8. Should be set to 0 when not using the piecewise linear algorithm.

Definition at line 255 of file plugin/pa-conversions/pa_curve_types_efr32.h

min

RAIL_TxPowerLevel_t RAIL_PaDescriptor_t::min

Min power level for this PA.

Definition at line 257 of file plugin/pa-conversions/pa_curve_types_efr32.h

max

```
RAIL_TxPowerLevel_t RAIL_PaDescriptor_t::max
```

Max power level for this PA.

Definition at line 259 of file plugin/pa-conversions/pa_curve_types_efr32.h

conversion

```
RAIL_PowerConversion_t RAIL_PaDescriptor_t::conversion
```

Union containing a pointer to algorithm-specific conversion data.

Definition at line 273 of file plugin/pa-conversions/pa_curve_types_efr32.h

RAIL_TxPowerCurvesConfigAlt_t

More generic structure containing information about piecewise linear curves and mapping tables, instead of specific PA's.

Public Attributes

RAIL_PaDescriptor_t	curves The curves for each PA.
uint32_t	signature Signature used for validation of the curves configuruation.
uint16_t	paVoltage PA VDD voltage, in millivolts.

Public Attribute Documentation

curves

RAIL_PaDescriptor_t RAIL_TxPowerCurvesConfigAlt_t::curves[RAIL_NUM_PA]

The curves for each PA.

Definition at line 284 of file `plugin/pa-conversions/pa_curve_types_efr32.h`

signature

uint32_t RAIL_TxPowerCurvesConfigAlt_t::signature

Signature used for validation of the curves configuruation.

Definition at line 286 of file `plugin/pa-conversions/pa_curve_types_efr32.h`

paVoltage

uint16_t RAIL_TxPowerCurvesConfigAlt_t::paVoltage

PA VDD voltage, in millivolts.

Definition at line 288 of file `plugin/pa-conversions/pa_curve_types_efr32.h`

Slxx3xx

Slxx3xx

Types specific to the Slxx3xx for dealing with the on-chip PAs.

Macros

- #define

RAIL_POWER_MODE_IS_ANY_DBM_POWERSETTING_MAPPING_TABLE (x)

The number of PA's on this chip (including Virtual PAs).
- #define

RAIL_POWER_MODE_SUPPORTS_RAW_SETTING (x)

Convenience macro to check if the power mode supports raw setting.

TrustZone

TrustZone

APIs to use RAIL with TrustZone enabled and peripherals configured as secure.

RAIL internally accesses CMU, EMU, GPIO, LDMA, HFXO, PRS and SYSRTC. If some of them are configured as secure peripherals, some RAIL code must be executed as secure code. To do so, callbacks gathered in `RAIL_TZ_Config_t` must be implemented and passed to RAIL through `RAIL_TZ_InitNonSecure()`. Each callback must do the non-secure/secure transition, call `RAIL_TZ_CheckPeripheralsSecureStates()` and then call the corresponding RAIL API from secure world:

Secure peripheral	Callbacks to implement
CMU	<code>RAIL_TZ_Config_t::changedDcdcCallback</code> , <code>RAIL_TZ_Config_t::configAntennaGpioCallback</code> , <code>RAIL_TZ_Config_t::radioClockEnableCallback</code> , <code>RAIL_TZ_Config_t::getRadioClockFreqHzCallback</code> , <code>RAIL_TZ_Config_t::rfecaClockEnableCallback</code> , <code>RAIL_TZ_Config_t::rfecalsClockEnabledCallback</code>
EMU	<code>RAIL_TZ_Config_t::readInternalTemperatureCallback</code> , <code>RAIL_TZ_Config_t::enableSecureRadioIrqsCallback</code> , <code>RAIL_TZ_Config_t::disableSecureRadioIrqsCallback</code>
GPIO	<code>RAIL_TZ_Config_t::configAntennaGpioCallback</code>
LDMA	<code>RAIL_TZ_Config_t::radioPerformM2mLdmaCallback</code>
HFXO	<code>RAIL_TZ_Config_t::configureHfxoCallback</code>

RAIL internally calls platform functions that access CMU, EMU, GPIO, LDMA HFXO and PRS. If some of them are configured as secure peripherals, some functions must be executed as secure code. To do so, those functions are prepended with weak symbols and must be overwritten to do the non-secure/secure transition and call the corresponding platform function from secure world:

Secure peripheral	Platform functions to overwrite
CMU	<code>CMU_ClockEnable</code> , <code>EMU_DCDCSetPFMXTIMEOUTMaxCtrl</code> (DCDC access: <code>SL_TRUSTZONE_PERIPHERAL_DCDC_S</code>)
HFXO	<code>CMU_HFXOCTuneSet</code> , <code>CMU_HFXOCTuneGet</code> , <code>CMU_HFXOCOREBiasCurrentCalibrate</code>
PRS	<code>PRS_SourceAsyncSignalSet</code> , <code>PRS_GetFreeChannel</code> , <code>PRS_ConnectConsumer</code> , <code>PRS_PinOutput</code> , <code>PRS_Combine</code>

When there is a combination of secure and non-secure peripherals, defines must be added in secure application slcp file so non-secure peripherals can properly accessed by secure code. Example with only CMU non-secure:

```
define:
- name: SL_TRUSTZONE_PERIPHERAL_CMU_S
  value: 0
  condition: [trustzone_secure]
- name: SL_TRUSTZONE_PERIPHERAL_EMU_S
  value: 1
  condition: [trustzone_secure]
- name: SL_TRUSTZONE_PERIPHERAL_GPIO_S
  value: 1
  condition: [trustzone_secure]
- name: SL_TRUSTZONE_PERIPHERAL_LDMA_S
  value: 1
  condition: [trustzone_secure]
- name: SL_TRUSTZONE_PERIPHERAL_HFXO0_S
  value: 1
```

```

condition:[trustzone_secure]- name: SL_TRUSTZONE_PERIPHERAL_EMU_S
value:1
condition:[trustzone_secure]- name: SL_TRUSTZONE_PERIPHERAL_GPIO_S
value:1
condition:[trustzone_secure]- name: SL_TRUSTZONE_PERIPHERAL_LDMA_S
value:1
condition:[trustzone_secure]- name: SL_TRUSTZONE_PERIPHERAL_HFXO0_S
value:1
condition:[trustzone_secure]- name: SL_TRUSTZONE_PERIPHERAL_PRS_S
value:1
condition:[trustzone_secure]- name: SL_TRUSTZONE_PERIPHERAL_SYSRTC_S
value:1
condition:[trustzone_secure]

```

Some RAIL API are not supported with EMU, GPIO, LDMA, HFXO, PRS or SYSRTC configured secure:

Secure peripheral	Unsupported RAIL API/features
EMU	RAIL_StartThermistorMeasurement() , RAIL_InitPowerManager()
GPIO	RAIL_EnableDirectMode() , RAIL_EnableDirectModeAlt() , RAIL_EnablePti() , RAIL_ConfigPti() , RAIL_ConfigHFXOThermistor() , RAIL_StartThermistorMeasurement() , RAIL_ConfigVdet() , RAIL_BLE_ConfigAoxAntenna()
LDMA	RAIL_IEEE802154_SUPPORTS_RX_CHANNEL_SWITCHING (RAIL_IEEE802154_ConfigRxChannelSwitching() and RAIL_RX_OPTION_CHANNEL_SWITCHING)
HFXO	RAIL_StartThermistorMeasurement() , RAIL_EnableVdet() , RAIL_GetVdet()
PRS	RAIL_EnablePrsLnaBypass()
SYSRTC	RAIL_ConfigSleep() with RAIL_SleepConfig_t::RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED , RAIL_ConfigSleepAlt() with RAIL_SleepConfig_t::RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED

Modules

[RAIL_TZ_Config_t](#)

Typedefs

```

typedef RAIL\_TZ\_ChangedDcdcCallbackPtr\_t(void)
RAIL\_Status\_t(*
    A pointer to the callback used to switch to secure world and run RAIL\_ChangedDcdc\(\).

typedef RAIL\_TZ\_ConfigAntennaGpioCallbackPtr\_t(const RAIL\_AntennaConfig\_t *config)
RAIL\_Status\_t(*
    A pointer to the callback used to switch to secure world and run RAIL\_TZ\_ConfigAntennaGpio\(\).

typedef void(*) RAIL\_TZ\_RadioClockEnableCallbackPtr\_t(void)
    A pointer to the callback used to switch to secure world and run RAIL\_TZ\_RadioClockEnable\(\).

typedef RAIL\_TZ\_GetRadioClockFreqHzCallbackPtr\_t(void)
uint32_t(*)
    A pointer to the callback used to switch to secure world and run RAIL\_GetRadioClockFreqHz\(\).

typedef void(*) RAIL\_TZ\_RfecaClockEnableCallbackPtr\_t(void)
    A pointer to the callback used to switch to secure world and run RAIL\_TZ\_RfecaClockEnable\(\).

typedef bool(*) RAIL\_TZ\_RfecalsClockEnabledCallbackPtr\_t(void)
    A pointer to the callback used to switch to secure world and run RAIL\_TZ\_RfecalsClockEnabled\(\).

typedef RAIL\_TZ\_ReadInternalTemperatureCallbackPtr\_t(uint16_t *internalTemperatureKelvin, bool
RAIL\_Status\_t(*
    enableTemperatureInterrupts)
    A pointer to the callback used to switch to secure world and run RAIL\_TZ\_ReadInternalTemperature\(\).

typedef void(*) RAIL\_TZ\_EnableSecureRadioIrqsCallbackPtr\_t(void)
    A pointer to the callback used to switch to secure world and run RAIL\_TZ\_EnableSecureRadioIrqs\(\).

```

typedef void(*	RAIL_TZ_DisableSecureRadioIrqsCallbackPtr_t (void)	A pointer to the callback used to switch to secure world and run RAIL_TZ_DisableSecureRadioIrqs() .
typedef RAIL_Status_t (*	RAIL_TZ_RadioPerformM2mLdmaCallbackPtr_t (uint32_t *pDest, const uint32_t *pSrc, uint32_t numWords)	A pointer to the callback used to switch to secure world and run RAIL_TZ_RadioPerformM2mLdma() .
typedef RAIL_Status_t (*	RAIL_TZ_ConfigureHfxoCallbackPtr_t (void)	A pointer to the callback used to switch to secure world and run RAIL_TZ_ConfigureHfxo() .

Functions

RAIL_Status_t	RAIL_TZ_InitNonSecure (const RAIL_TZ_Config_t *pTzConfig)	Init RAIL TrustZone feature for non-secure world.
RAIL_Status_t	RAIL_TZ_InitSecure (void)	Init RAIL TrustZone feature for secure world.
RAIL_Status_t	RAIL_TZ_CheckPeripheralsSecureStates (void)	Check the secure state of peripherals used by RAIL.
RAIL_Status_t	RAIL_TZ_RadioClockEnable (void)	Enable radio clocks.
RAIL_Status_t	RAIL_TZ_RfecaClockEnable (void)	Enable RFECA clocks.
bool	RAIL_TZ_RfecalsClockEnabled (void)	Indicate whether RFECA clocks are enabled.
RAIL_Status_t	RAIL_TZ_ReadInternalTemperature (uint16_t *internalTemperatureKelvin, bool enableTemperatureInterrupts)	Read the internal temperature.
RAIL_Status_t	RAIL_TZ_EnableSecureRadioIrqs (void)	Enable secure peripheral interrupts needed by the radio.
RAIL_Status_t	RAIL_TZ_DisableSecureRadioIrqs (void)	Disable secure peripheral interrupts needed by the radio.
RAIL_Status_t	RAIL_TZ_RadioPerformM2mLdma (uint32_t *pDest, const uint32_t *pSrc, uint32_t numWords)	Perform Idma transfer for the radio.
RAIL_Status_t	RAIL_TZ_ConfigureHfxo (void)	Configure HFXO.
RAIL_Status_t	RAIL_TZ_ConfigAntennaGpio (const RAIL_AntennaConfig_t *config)	Set GPIO for antenna config.

Typedef Documentation

RAIL_TZ_ChangedDcdcCallbackPtr_t

[RAIL_TZ_ChangedDcdcCallbackPtr_t](#))(void)

A pointer to the callback used to switch to secure world and run [RAIL_ChangedDcdc\(\)](#).

Returns

- Status code indicating success of the function call.

Definition at line 5966 of file `common/rail_types.h`

RAIL_TZ_ConfigAntennaGpioCallbackPtr_t

```
RAIL_TZ_ConfigAntennaGpioCallbackPtr_t )(const RAIL_AntennaConfig_t *config)
```

A pointer to the callback used to switch to secure world and run [RAIL_TZ_ConfigAntennaGpio\(\)](#).

Parameters

[in]	config	A pointer to a configuration structure applied to the relevant Antenna Configuration registers. A NULL configuration will produce undefined behavior.

Returns

- Status code indicating success of the function call.

Definition at line 5978 of file `common/rail_types.h`

RAIL_TZ_RadioClockEnableCallbackPtr_t

```
RAIL_TZ_RadioClockEnableCallbackPtr_t )(void)
```

A pointer to the callback used to switch to secure world and run [RAIL_TZ_RadioClockEnable\(\)](#).

Definition at line 5986 of file `common/rail_types.h`

RAIL_TZ_GetRadioClockFreqHzCallbackPtr_t

```
RAIL_TZ_GetRadioClockFreqHzCallbackPtr_t )(void)
```

A pointer to the callback used to switch to secure world and run [RAIL_GetRadioClockFreqHz\(\)](#).

Returns

- Radio subsystem clock frequency in Hz.

Definition at line 5996 of file `common/rail_types.h`

RAIL_TZ_RfecaClockEnableCallbackPtr_t

```
RAIL_TZ_RfecaClockEnableCallbackPtr_t )(void)
```

A pointer to the callback used to switch to secure world and run [RAIL_TZ_RfecaClockEnable\(\)](#).

Definition at line 6004 of file `common/rail_types.h`

RAIL_TZ_RfecalsClockEnabledCallbackPtr_t

```
RAIL_TZ_RfecalsClockEnabledCallbackPtr_t )(void)
```

A pointer to the callback used to switch to secure world and run [RAIL_TZ_RfecalsClockEnabled\(\)](#).

Returns

- true if RFECA clocks are enabled; false otherwise

Definition at line 6014 of file common/rail_types.h

RAIL_TZ_ReadInternalTemperatureCallbackPtr_t

```
RAIL_TZ_ReadInternalTemperatureCallbackPtr_t )(uint16_t *internalTemperatureKelvin, bool enableTemperatureInterrupts)
```

A pointer to the callback used to switch to secure world and run [RAIL_TZ_ReadInternalTemperature\(\)](#).

Parameters

[out]	internalTemperatureKelvin	A pointer to the internal temperature in Kelvin.
[in]	enableTemperatureInterrupts	Indicate whether temperature interrupts are enabled.

Returns

- Status code indicating success of the function call.

Definition at line 6028 of file common/rail_types.h

RAIL_TZ_EnableSecureRadioIrqsCallbackPtr_t

```
RAIL_TZ_EnableSecureRadioIrqsCallbackPtr_t )(void)
```

A pointer to the callback used to switch to secure world and run [RAIL_TZ_EnableSecureRadioIrqs\(\)](#).

Definition at line 6037 of file common/rail_types.h

RAIL_TZ_DisableSecureRadioIrqsCallbackPtr_t

```
RAIL_TZ_DisableSecureRadioIrqsCallbackPtr_t )(void)
```

A pointer to the callback used to switch to secure world and run [RAIL_TZ_DisableSecureRadioIrqs\(\)](#).

Definition at line 6045 of file common/rail_types.h

RAIL_TZ_RadioPerformM2mLdmaCallbackPtr_t

```
RAIL_TZ_RadioPerformM2mLdmaCallbackPtr_t )(uint32_t *pDest, const uint32_t *pSrc, uint32_t numWords)
```

A pointer to the callback used to switch to secure world and run [RAIL_TZ_RadioPerformM2mLdma\(\)](#).

Parameters

[in]	pDest	A pointer to the destination data.
[in]	pSrc	A pointer to the source data.
[in]	numWords	Number of words to transfer.

Returns

- Status code indicating success of the function call.

Definition at line 6058 of file common/rail_types.h

RAIL_TZ_ConfigureHfxoCallbackPtr_t

```
RAIL_TZ_ConfigureHfxoCallbackPtr_t )(void)
```

A pointer to the callback used to switch to secure world and run [RAIL_TZ_ConfigureHfxo\(\)](#).

Definition at line 6068 of file common/rail_types.h

Function Documentation

RAIL_TZ_InitNonSecure

```
RAIL_Status_t RAIL_TZ_InitNonSecure (const RAIL\_TZ\_Config\_t * pTzConfig)
```

Init RAIL TrustZone feature for non-secure world.

Parameters

[in]	pTzConfig	A non-NULL pointer to a RAIL_TZ_Config_t structure.

Returns

- Status code indicating success of the function call.

Note

- This function must only be called from non-secure world (only if TrustZone is activated) on platforms having [RAIL_SUPPORTS_TRUSTZONE_SECURE_PERIPHERALS](#). It must be called before [RAIL_Init\(\)](#) and it must be called again with updated [RAIL_TZ_Config_t](#) if peripherals secure configuration has changed.

Definition at line 6955 of file common/rail.h

RAIL_TZ_InitSecure

```
RAIL_Status_t RAIL_TZ_InitSecure (void )
```

Init RAIL TrustZone feature for secure world.

Parameters

N/A		

Returns

- Status code indicating success of the function call.

Note

- This function must only be called from secure world (only if TrustZone is activated) on platforms having [RAIL_SUPPORTS_TRUSTZONE_SECURE_PERIPHERALS](#). It must be called before starting the non-secure application.

Definition at line 6967 of file common/rail.h

RAIL_TZ_CheckPeripheralsSecureStates

RAIL_Status_t RAIL_TZ_CheckPeripheralsSecureStates (void)

Check the secure state of peripherals used by RAIL.

Parameters

N/A		

Returns

- Status code indicating success of the function call.

Note

- This function must only be called from secure world and it must be called at the beginning of each RAIL TrustZone callbacks ([RAIL_TZ_Config_t](#)) secure code to avoid secure fault.

Definition at line 6978 of file common/rail.h

RAIL_TZ_RadioClockEnable

RAIL_Status_t RAIL_TZ_RadioClockEnable (void)

Enable radio clocks.

Parameters

N/A		

Returns

- Status code indicating success of the function call.

Note

- This function must only be called from secure world when CMU is configured as secure TrustZone peripheral.

Definition at line 6989 of file common/rail.h

RAIL_TZ_RfecaClockEnable

RAIL_Status_t RAIL_TZ_RfecaClockEnable (void)

Enable RFECA clocks.

Parameters

N/A		

Returns

- Status code indicating success of the function call.

Note

-

This function must only be called from secure world when CMU is configured as secure TrustZone peripheral.

Definition at line 7000 of file common/rail.h

RAIL_TZ_RfecalsClockEnabled

```
bool RAIL_TZ_RfecalsClockEnabled (void )
```

Indicate whether RFECA clocks are enabled.

Parameters

N/A		

Returns

- true if RFECA clocks are enabled; false otherwise

Note

- This function must only be called from secure world when CMU is configured as secure TrustZone peripheral.

Definition at line 7011 of file common/rail.h

RAIL_TZ_ReadInternalTemperature

```
RAIL_Status_t RAIL_TZ_ReadInternalTemperature (uint16_t * internalTemperatureKelvin, bool enableTemperatureInterrupts)
```

Read the internal temperature.

Parameters

[out]	internalTemperatureKelvin	A pointer to the internal temperature in Kelvin.
[in]	enableTemperatureInterrupts	Indicate whether temperature interrupts are enabled.

Returns

- Status code indicating success of the function call.

Note

- This function must only be called from secure world when EMU is configured as secure TrustZone peripheral.

Definition at line 7026 of file common/rail.h

RAIL_TZ_EnableSecureRadiolrqs

```
RAIL_Status_t RAIL_TZ_EnableSecureRadiolrqs (void )
```

Enable secure peripheral interrupts needed by the radio.

Parameters

N/A		

Returns

- Status code indicating success of the function call.

Note

- This function must only be called from secure world when EMU is configured as secure TrustZone peripheral.

Definition at line 7038 of file common/rail.h

RAIL_TZ_DisableSecureRadiolrqs

RAIL_Status_t RAIL_TZ_DisableSecureRadiolrqs (void)

Disable secure peripheral interrupts needed by the radio.

Parameters

N/A		

Returns

- Status code indicating success of the function call.

Note

- This function must only be called from secure world when EMU is configured as secure TrustZone peripheral.

Definition at line 7049 of file common/rail.h

RAIL_TZ_RadioPerformM2mLdma

RAIL_Status_t RAIL_TZ_RadioPerformM2mLdma (uint32_t * pDest, const uint32_t * pSrc, uint32_t numWords)

Perform ldma transfer for the radio.

Parameters

[in]	pDest	A pointer to the destination data.
[in]	pSrc	A pointer to the source data.
[in]	numWords	Number of words to transfer.

Returns

- Status code indicating success of the function call.

Note

- This function must only be called from secure world when LDMA is configured as secure TrustZone peripheral.

Definition at line 7063 of file common/rail.h

RAIL_TZ_ConfigureHfxo

RAIL_Status_t RAIL_TZ_ConfigureHfxo (void)

Configure HFXO.

Parameters

N/A		

Returns

- Status code indicating success of the function call.

Note

- This function must only be called from secure world when HFXO is configured as secure TrustZone peripheral.

Definition at line 7076 of file common/rail.h

RAIL_TZ_ConfigAntennaGpio

```
RAIL_Status_t RAIL_TZ_ConfigAntennaGpio (const RAIL_AntennaConfig_t * config)
```

Set GPIO for antenna config.

Parameters

[in]	config	A pointer to a configuration structure applied to the relevant Antenna Configuration registers. A NULL configuration will produce undefined behavior.

Returns

- Status code indicating success of the function call.

Note

- This function must only be called from secure world when CMU or GPIO are configured as secure TrustZone peripheral.

Definition at line 7089 of file common/rail.h

RAIL_TZ_Config_t

Gather RAIL TrustZone callbacks pointers and booleans indicating peripheral secure configuration.

Public Attributes

RAIL_TZ_ChangedDcdcCallbackPtr_t	changedDcdcCallback See RAIL_TZ_ChangedDcdcCallbackPtr_t .
RAIL_TZ_ConfigAntennaGpioCallbackPtr_t	configAntennaGpioCallback See RAIL_TZ_ConfigAntennaGpioCallbackPtr_t .
RAIL_TZ_RadioClockEnableCallbackPtr_t	radioClockEnableCallback See RAIL_TZ_RadioClockEnableCallbackPtr_t .
RAIL_TZ_GetRadioClockFreqHzCallbackPtr_t	getRadioClockFreqHzCallback See RAIL_TZ_GetRadioClockFreqHzCallbackPtr_t .
RAIL_TZ_RfecaClockEnableCallbackPtr_t	rfecaClockEnableCallback See RAIL_TZ_RfecaClockEnableCallbackPtr_t .
RAIL_TZ_RfecalsClockEnabledCallbackPtr_t	rfecalsClockEnabledCallback See RAIL_TZ_RfecalsClockEnabledCallbackPtr_t .
RAIL_TZ_ReadInternalTemperatureCallbackPtr_t	readInternalTemperatureCallback See RAIL_TZ_ReadInternalTemperatureCallbackPtr_t .
RAIL_TZ_EnableSecureRadioIrqsCallbackPtr_t	enableSecureRadioIrqsCallback See RAIL_TZ_EnableSecureRadioIrqsCallbackPtr_t .
RAIL_TZ_DisableSecureRadioIrqsCallbackPtr_t	disableSecureRadioIrqsCallback See RAIL_TZ_DisableSecureRadioIrqsCallbackPtr_t .
RAIL_TZ_RadioPerformM2mLdmaCallbackPtr_t	radioPerformM2mLdmaCallback See RAIL_TZ_RadioPerformM2mLdmaCallbackPtr_t .
RAIL_TZ_ConfigureHfxoCallbackPtr_t	configureHfxoCallback See RAIL_TZ_ConfigureHfxoCallbackPtr_t .
bool	isCmuSecure Indicate whether CMU is configured as secure peripheral.
bool	isEmuSecure Indicate whether EMU is configured as secure peripheral.
bool	isGpioSecure Indicate whether GPIO is configured as secure peripheral.
bool	isLdmaSecure Indicate whether LDMA is configured as secure peripheral.

bool	isHfxoSecure	Indicate whether HFXO is configured as secure peripheral.
bool	isPrsSecure	Indicate whether PRS is configured as secure peripheral.
bool	isSysrtcSecure	Indicate whether SYSRTC is configured as secure peripheral.

Public Attribute Documentation

changedDcdcCallback

```
RAIL_TZ_ChangedDcdcCallbackPtr_t RAIL_TZ_Config_t::changedDcdcCallback
```

See [RAIL_TZ_ChangedDcdcCallbackPtr_t](#).

In non-secure world, it must be NULL if CMU is a non-secure peripheral.

Definition at line 6080 of file common/rail_types.h

configAntennaGpioCallback

```
RAIL_TZ_ConfigAntennaGpioCallbackPtr_t RAIL_TZ_Config_t::configAntennaGpioCallback
```

See [RAIL_TZ_ConfigAntennaGpioCallbackPtr_t](#).

In non-secure world, it must be NULL if CMU and GPIO are non-secure peripherals.

Definition at line 6086 of file common/rail_types.h

radioClockEnableCallback

```
RAIL_TZ_RadioClockEnableCallbackPtr_t RAIL_TZ_Config_t::radioClockEnableCallback
```

See [RAIL_TZ_RadioClockEnableCallbackPtr_t](#).

In non-secure world, it must be NULL if CMU is a non-secure peripheral.

Definition at line 6091 of file common/rail_types.h

getRadioClockFreqHzCallback

```
RAIL_TZ_GetRadioClockFreqHzCallbackPtr_t RAIL_TZ_Config_t::getRadioClockFreqHzCallback
```

See [RAIL_TZ_GetRadioClockFreqHzCallbackPtr_t](#).

In non-secure world, it must be NULL if CMU is a non-secure peripheral.

Definition at line 6096 of file common/rail_types.h

rfecaClockEnableCallback

```
RAIL_TZ_RfecaClockEnableCallbackPtr_t RAIL_TZ_Config_t::rfecaClockEnableCallback
```

See [RAIL_TZ_RfecaClockEnableCallbackPtr_t](#).

In non-secure world, it must be NULL if CMU is a non-secure peripheral.

Definition at line 6101 of file `common/rail_types.h`

rfecalsClockEnabledCallback

```
RAIL_TZ_RfecalsClockEnabledCallbackPtr_t RAIL_TZ_Config_t::rfecalsClockEnabledCallback
```

See [RAIL_TZ_RfecalsClockEnabledCallbackPtr_t](#).

In non-secure world, it must be NULL if CMU is a non-secure peripheral.

Definition at line 6106 of file `common/rail_types.h`

readInternalTemperatureCallback

```
RAIL_TZ_ReadInternalTemperatureCallbackPtr_t RAIL_TZ_Config_t::readInternalTemperatureCallback
```

See [RAIL_TZ_ReadInternalTemperatureCallbackPtr_t](#).

In non-secure world, it must be NULL if EMU is a non-secure peripheral.

Definition at line 6111 of file `common/rail_types.h`

enableSecureRadioIrqsCallback

```
RAIL_TZ_EnableSecureRadioIrqsCallbackPtr_t RAIL_TZ_Config_t::enableSecureRadioIrqsCallback
```

See [RAIL_TZ_EnableSecureRadioIrqsCallbackPtr_t](#).

In non-secure world, it must be NULL if EMU is a non-secure peripheral.

Definition at line 6116 of file `common/rail_types.h`

disableSecureRadioIrqsCallback

```
RAIL_TZ_DisableSecureRadioIrqsCallbackPtr_t RAIL_TZ_Config_t::disableSecureRadioIrqsCallback
```

See [RAIL_TZ_DisableSecureRadioIrqsCallbackPtr_t](#).

In non-secure world, it must be NULL if EMU is a non-secure peripheral.

Definition at line 6121 of file `common/rail_types.h`

radioPerformM2mLdmaCallback


```
RAIL_TZ_RadioPerformM2mLdmaCallbackPtr_t RAIL_TZ_Config_t::radioPerformM2mLdmaCallback
```

See [RAIL_TZ_RadioPerformM2mLdmaCallbackPtr_t](#).

In non-secure world, it must be NULL if LDMA is a non-secure peripheral or if RAIL must not use LDMA.

Definition at line 6127 of file `common/rail_types.h`

configureHfxoCallback

```
RAIL_TZ_ConfigureHfxoCallbackPtr_t RAIL_TZ_Config_t::configureHfxoCallback
```

See [RAIL_TZ_ConfigureHfxoCallbackPtr_t](#).

In non-secure world, it must be NULL if HFXO is a non-secure peripheral.

Definition at line 6132 of file `common/rail_types.h`

isCmuSecure

```
bool RAIL_TZ_Config_t::isCmuSecure
```

Indicate whether CMU is configured as secure peripheral.

Definition at line 6136 of file `common/rail_types.h`

isEmuSecure

```
bool RAIL_TZ_Config_t::isEmuSecure
```

Indicate whether EMU is configured as secure peripheral.

Definition at line 6140 of file `common/rail_types.h`

isGpioSecure

```
bool RAIL_TZ_Config_t::isGpioSecure
```

Indicate whether GPIO is configured as secure peripheral.

Definition at line 6144 of file `common/rail_types.h`

isLdmaSecure

```
bool RAIL_TZ_Config_t::isLdmaSecure
```

Indicate whether LDMA is configured as secure peripheral.

Definition at line 6148 of file common/rail_types.h

isHfxoSecure

```
bool RAIL_TZ_Config_t::isHfxoSecure
```

Indicate whether HFXO is configured as secure peripheral.

Definition at line 6152 of file common/rail_types.h

isPrsSecure

```
bool RAIL_TZ_Config_t::isPrsSecure
```

Indicate whether PRS is configured as secure peripheral.

Definition at line 6156 of file common/rail_types.h

isSysrtcSecure

```
bool RAIL_TZ_Config_t::isSysrtcSecure
```

Indicate whether SYSRTC is configured as secure peripheral.

Definition at line 6160 of file common/rail_types.h

Components

Components

Software Components

These are some of the RAIL software components available for application use.

Libraries

RAIL library software components provide the ability to choose support for single or multiple protocols. Current components include:

- [RAIL Library, Single Protocol](#)
- [RAIL Library, Multiprotocol](#)

Utilities

Radio utility software components provide optional source code that aid with radio setup and functionality. Current components include:

- [Angle of Arrival/Angle of Departure \(AoX\)](#)
- [Antenna Diversity](#)
- [BLE Channel Sounding \(CS\)](#)
- [BLE Channel Sounding \(CS\) Antenna Offset](#)
- [Built-in PHYs across HFXO frequencies](#)
- [Callbacks](#)
- [Coexistence](#)
- [Direct Memory Access \(DMA\)](#)
- [Voltage Detector \(VDET\) for Front End Modules](#)
- [Front End Module \(FEM\)](#)
- [Device Information Utility](#)
- [Initialization](#)
- [Packet Trace Interface \(PTI\)](#)
- [Power Amplifier \(PA\)](#)
- [PA power configs in persistent memory \(EFR32xG24/xG26\)](#)
- [Protocol](#)
- [Recommended](#)
- [RF Path](#)
- [RF Path Switch Utility](#)
- [Received Signal Strength Indicator \(RSSI\)](#)
- [Radio Sequencer Image Selection Utility](#)
- [Software Modem \(SFM\) sequencer image selection Utility](#)
- [Thermistor Utility](#)

RAILtest Application

RAILtest application software components are specific to the RAILtest application. Current components include:

- [RAILtest Application Core](#)
- [RAILtest Application Graphics](#)

Modules

[RAIL Library](#)

[Angle of Arrival/Departure \(AoX\) Utility](#)

[Antenna Diversity Utility](#)

[BLE Channel Sounding \(CS\) Utility](#)

[BLE Channel Sounding \(CS\) Antenna Offset Utility](#)

[Built-in PHYs for alternate HFXO frequencies](#)

[Callbacks Utility](#)

[Coexistence Utility](#)

[Direct Memory Access \(DMA\) Utility](#)

[Front End Module Voltage Detection \(VDET\) Utility](#)

[Front End Module \(FEM\) Utility](#)

[Initialization Utility](#)

[Packet Trace Interface \(PTI\) Utility](#)

[Power Amplifier \(PA\) Utility](#)

[Protocol Utility](#)

[Recommended Utility](#)

[RF Path Utility](#)

[Received Signal Strength Indicator \(RSSI\) Utility](#)

[RAILtest Application Core](#)

[RAILtest Application Graphics](#)

[Thermistor Utility](#)

[RF Path Switch Utility](#)

[Radio Sequencer Image Selection Utility](#)

[Software Modem \(SFM\) sequencer image selection Utility](#)

RAIL Library

RAIL Library

RAIL Multiprotocol Library

The RAIL multiprotocol library allows multiple protocols to be initialized, set up, and managed at the library level. Dynamic Multiprotocol (DMP) requires the use of the RAIL multiprotocol library. Additional information can be found here: [RAIL Multiprotocol](#).

RAIL Single Protocol Library

The RAIL single protocol library reduces flash and memory footprints compared to the RAIL multiprotocol library. The [RAIL_SchedulerInfo_t](#) input parameter, used by select RAIL library API functions, is always ignored in the RAIL single protocol library. The single protocol library can be used for Switched Multiprotocol (SMP) because the RAIL library is initialized and used for one wireless protocol at a time.

Angle of Arrival/Departure (AoX) Utility

Angle of Arrival/Departure (AoX) Utility

RAIL Angle of Arrival/Departure (AoX) Utility

This optional software component can be enabled to include default functionality related to AoX configuration. AoX allows the Bluetooth stack to estimate the relative angle of another radio based on samples from multiple antennas.

The RAIL AoX Utility is only necessary if 2 or more antennas are configured. If fewer than 2 antennas are configured, CTE(constant tone extension) data can be transmitted or received on the default antenna.

Configuration Options

The following configuration options can be changed:

- Number of AoX antenna pins(0-6)

The following hardware options can be changed:

- GPIO pin for each antenna

Antenna Diversity Utility

Antenna Diversity Utility

RAIL Antenna Diversity Utility

This optional software component can be enabled to include default functionality related to antenna diversity configuration.

When using a Silicon Labs-developed board, the configuration options for this software component are set up automatically based on the selected board. When using a custom board, manually configure the configuration options.

Configuration Options

The following configuration options can be changed:

- Antenna Diversity:
 - RX Antenna Diversity Mode: Choose the mode to use for receive antenna diversity. The possible modes are disabled, enabled, Antenna 0 only, or Antenna 1 only.
 - TX Antenna Diversity Mode: Choose the mode to use for transmit antenna diversity. The possible modes are disabled, enabled, Antenna 0 only, or Antenna 1 only.
 - Enable/disable Runtime PHY Selection: This option only works when using the Silicon Labs Zigbee or Open Thread stacks. If enabled it will allow turning on and off the antenna diversity version of the IEEE 802.15.4 PHY at runtime by these stacks.

The following hardware options can be changed:

- Configure the antenna diversity transmit select GPIO (ANT0 signal).
- Configure the antenna diversity inverse transmit select GPIO (ANT1 signal).

BLE Channel Sounding (CS) Utility

BLE Channel Sounding (CS) Utility

RAIL BLE Channel Sounding Utility

This optional software component can be enabled to include default functionality related to CS configurations performed by the Bluetooth stack. For a detailed overview of CS and relevant BLE documentation, see the following:

[Channel Sounding](#)

[Bluetooth Documentation](#)

BLE Channel Sounding (CS) Antenna Offset Utility

BLE Channel Sounding (CS) Antenna Offset Utility

RAIL BLE Channel Sounding Antenna Offset Utility

This optional software component can be enabled to include board-specific antenna offset values which are accounted for during CS distance measurements. The maximum number of antennas supported for CS is 4, with each antenna carrying a distance offset value in units of centimeters for both a wireless and wired antenna configuration.

Configuration Options

The following configuration macros are available for modification:

- CS Antenna Offset:
 - `SL_RAIL_UTIL_CS_ANTENNA_COUNT`: Number of antennas used during CS.
 - `SL_RAIL_UTIL_CS_ANTENNA_CONFIG_TYPE`: Indicates the antenna configuration type being used, either wireless or wired. By default, a wireless configuration will be selected, with the macro set to `SL_RAIL_UTIL_CS_ANTENNA_OFFSET_CONFIG_TYPE_WIRELESS`, or 0. To select a wired antenna configuration type, set the macro field to `SL_RAIL_UTIL_CS_ANTENNA_OFFSET_CONFIG_TYPE_WIRED`, or 1.
 - `SL_RAIL_UTIL_CS_ANTENNA_OFFSET_WIRELESS_CM`: Wireless antenna offset values for each antenna, in cm.
 - `SL_RAIL_UTIL_CS_ANTENNA_OFFSET_WIRED_CM`: Wired antenna offset values for each antenna, in cm.

Note: The antenna offset values for each antenna are calculated offsets for each antenna and should be used. Additionally, the number of antenna offsets set for a wireless or wired antenna configuration must be proportional or greater than the number of antennas configured for CS. Offset values for unused antennas will be ignored.

Built-in PHYs for alternate HFXO frequencies

Built-in PHYs for alternate HFXO frequencies

RAIL Utility for built-in PHYs

This optional software component can be enabled to override the built-in PHYs used to support standards-based protocols based on the HFXO frequency configured as part of the `device_init_hfxo` component.

Currently the following combinations of chips and frequencies are supported:

- EFR32XG24: 38.4 MHz, 39 MHz and 40 MHz
- EFR32XG26: 38.4 MHz, 39 MHz and 40 MHz

On unsupported chips, this component is safe to include, and will have no effect.

Callbacks Utility

Callbacks Utility

RAIL Callbacks Utility

This optional software component can be enabled to include default functionality related to RAIL callback configuration.

This utility provides basic callbacks that can be made available to RAIL. If a callback is not registered for use, it will be compiled out. The provided callbacks include:

- [RAILCb_AssertFailed](#)
 - This callback is called when a RAIL library-level assertion occurs. This callback occurs regardless of the other configuration options enabled in this software component.
 - This callback overrides a weak function in the RAIL library and as such cannot be compiled out. Any assert within the RAIL library will cause this callback to run and provide information as to the cause.
 - The weak function `sl_rail_util_on_assert_failed` is provided for application use and will be called when [RAILCb_AssertFailed](#) is called.
 - This callback override can be configured to be excluded, in which case the application is responsible for creating its own [RAILCb_AssertFailed](#) override if so desired.
- `sl_rail_util_on_rf_ready`
 - This callback is called when the call to [RAIL_Init](#) completes.
 - The `Initialize RAIL` configuration option must be enabled, and this callback must be passed as a parameter to [RAIL_Init](#) to enable its use.
 - The weak function `sl_rail_util_on_rf_ready` is provided for application use and will be called when `sl_rail_util_on_rf_ready` is called.
- `sl_rail_util_on_channel_config_change`
 - This callback is called when an automatic switch from one channel configuration entry to another is performed internal to the RAIL library.
 - The `Initialize RAIL` and `Protocol Configuration` configuration options must be enabled, and this callback must be passed as a parameter to [RAIL_ConfigChannels](#) to enable its use.
 - The weak function `sl_rail_util_on_channel_config_change` is provided for application use and will be called when `sl_rail_util_on_channel_config_change` is called.
- `sl_rail_util_on_event`
 - This callback is called when a configured radio event occurs. The radio events capable of causing this callback are identified in [RAIL_Events_t](#).
 - The `Initialize RAIL` and `Radio Event Configuration` configuration options must be enabled, and this callback must be passed as a parameter to [RAIL_Init](#) to enable its use.
 - The weak function `sl_rail_util_on_event` is provided for application use and will be called when `sl_rail_util_on_event` is called.

Configuration Options

The following configuration options can be changed:

- Include/exclude [RAILCb_AssertFailed](#) callback override.

Coexistence Utility

Coexistence Utility

RAIL Coexistence Utility

This optional software component can be enabled to include default functionality related to the coexistence configuration.

See the following documents for in-depth explanations about how to configure and use the available coexistence features:

- AN1128: Bluetooth® Coexistence with Wi-Fi®
- AN1017: Zigbee® and Silicon Labs® Thread Coexistence with Wi-Fi®
- UG350: Silicon Labs Coexistence Development Kit (SLWSTK-COEXBP)

Direct Memory Access (DMA) Utility

Direct Memory Access (DMA) Utility

RAIL DMA Utility

This optional software component can be enabled to include default functionality related to DMA configuration.

Configuration Options

The following configuration options can be changed:

- Enable/disable a DMA channel for use by RAIL. Allocating a DMA channel to the RAIL library allows for faster radio initialization.
 - Allow for a DMA channel to be selected automatically.
 - If not automatically allocating a DMA channel, identify a specific channel to allocate.

Front End Module Voltage Detection (VDET) Utility

Front End Module Voltage Detection (VDET) Utility

RAIL VDET Utility

This optional software component can be enabled to include default functionality related to VDET configuration. VDET allows customers to measure a voltage during specific timestamps of a transmit packet. This information is used to tune FEM output power measurements during manufacturing and allows compensation for variability in the output path.

Configuration Options

- VDET mode
- VDET resolution
- VDET sample delay

The following hardware pin must be enabled to use VDET and is fixed to specific GPIO pins on supported EFR32 devices that cannot be changed.

- The RAC.AUXADC_IN8_SHIM_EN GPIO pin.

Front End Module (FEM) Utility

Front End Module (FEM) Utility

Radio FEM Utility

Enabling this optional software component includes default functionality related to FEM configuration. It also configures the GPIOs together with the Peripheral Reflex System (PRS) signals to control the FEM.

Configuration Options

The following options can be configured:

- Enable RX Mode: Enables the output of RX based on the selected port and pin.
- Enable TX Mode: Enables the output of TX based on the selected port and pin.
- Enable Bypass Mode: Enables communication that bypasses the PA (Power Amplifier) and the LNA (Low Noise Amplifier) on the FEM.
- Enable TX High Power Mode: Enables high power transmit mode on the FEM, if disabled low power transmit will be used.
- Enable automatic PRS LNA Bypass Mode: Enables dynamic LNA bypass when receiving frames with high power. This is available on platforms having [RAIL_SUPPORTS_PRS_LNA_BYPASS](#).
 - SL_FEM_UTIL_AUTO_PRS_LNA_BYPASS_THRESHOLD: see [RAIL_PrsLnaBypassConfig_t::threshold](#)
 - SL_FEM_UTIL_AUTO_PRS_LNA_BYPASS_DELTA_RSSI_DMB: see [RAIL_PrsLnaBypassConfig_t::deltaRssiDbm](#)
 - SL_FEM_UTIL_AUTO_PRS_LNA_BYPASS_POLARITY: see [RAIL_PrsLnaBypassConfig_t::polarity](#)
 - SL_FEM_UTIL_AUTO_PRS_LNA_BYPASS_TIMEOUT_US: see [RAIL_PrsLnaBypassConfig_t::timeoutUs](#)
 - SL_FEM_UTIL_AUTO_PRS_LNA_BYPASS_COMBINE: Demonstrate how to perform logical combination between dynamic LNA bypass PRS signal and another PRS signal.
 - SL_FEM_UTIL_AUTO_PRS_LNA_BYPASS_NO_COMBINE: Dynamic LNA bypass PRS signal directly routed to SL_FEM_UTIL_BYPASS_PORT/PIN.
 - SL_FEM_UTIL_AUTO_PRS_LNA_BYPASS_OR_PAEN_COMBINE: Dynamic LNA bypass PRS signal ored with PA enable on SL_FEM_UTIL_BYPASS_PRS_COMBINE channel and routed to SL_FEM_UTIL_BYPASS_PORT/PIN.
 - SL_FEM_UTIL_AUTO_PRS_LNA_BYPASS_XOR_LNAEN_COMBINE: Dynamic LNA bypass PRS signal xored with LNA enable on SL_FEM_UTIL_BYPASS_PRS_COMBINE channel and routed to SL_FEM_UTIL_BYPASS_PORT/PIN.

Typically, the signal routing is as follows:

- SL_FEM_UTIL_BYPASS: Bypass control (FEM pin **CPS**).
- SL_FEM_UTIL_RX: RX control (FEM pin **CRX**). Active based on the radio activity (whenever the LNA is enabled).
- SL_FEM_UTIL_SLEEP: Sleep control (FEM pin **CSD**). Active based on the radio activity (whenever the PA or LNA are enabled).
- SL_FEM_UTIL_TX: TX control (FEM pin **CTX**). Active based on the radio activity (whenever the PA is enabled).
- SL_FEM_UTIL_TX_HIGH_POWER: Tx power mode control (FEM pin **CHL**).

Refer to the FEM datasheet for the required settings as not all configuration options are applicable to every module.

Initialization Utility

Initialization Utility

RAIL Initialization Utility

This optional software component can be enabled to include default functionality related to RAIL initialization configuration.

Note: Multiple instances of this component can be enabled in application code. This is particularly useful for multiprotocol scenarios.

Configuration Options

The following configuration options can be changed:

- Enable/disable RAIL initialization.
 - Enable/disable Bluetooth BLE support. If not using Bluetooth, this option can be disabled to reduce memory usage, but if left enabled, no difference in performance occurs.
 - Enable/disable dynamic multiprotocol (DMP) support. This option allocates memory for use by the radio scheduler incorporated into the multiprotocol RAIL library. This should be left disabled for single protocol RAIL library use, which includes switched multiprotocol (SMP) support.
 - Enable/disable initialization complete callback.
- Enable/disable channel configuration.
 - Configure a proprietary channel configuration on boot. This specifically involves the use of the `channelConfigs` array in `rail_config.c/h`.
 - If using a proprietary channel configuration, specify the channel index for use (i.e., index `x` of array `channelConfigs[x]`).
- Enable/disable calibration configuration.
 - Enable/disable temperature-dependent calibrations (e.g., voltage controlled oscillator calibration).
 - Enable/disable one-time calibrations (e.g., image rejection calibration).
- Enable/disable auto transition configuration.
 - Configure TX transitions on Success and Failure.
 - Configure RX transitions on Success and Failure.
- Enable/disable radio callback event configuration.
 - Configure RX events.
 - Configure TX events.
 - Configure protocol-specific events.
 - Configure dynamic multiprotocol-specific events.
 - Configure maintenance events. Note: Some protocol-specific events share the same event flag with other protocols; enabling one will enable them all. See `rail_types.h::RAIL_Event_t`.

Packet Trace Interface (PTI) Utility

Packet Trace Interface (PTI) Utility

RAIL PTI Utility

This optional software component can be enabled to configure the PTI interface. The PTI interface allows the user to collect receive and transmit packet data in real time over a dedicated serial interface for debugging network activity.

When using a Silicon Labs radio board, the configuration options for this software component are set up automatically to match the pin mapping on that board. When using a custom board these options can be set any way that makes sense to the user. Keep in mind that the Silicon Labs WSTK is able to capture and parse this data but currently requires UART mode and defaults to a 1.6Mbps data rate.

Configuration Options

The following configuration options can be changed:

- PTI mode
 - [RAIL_PT_MODE_UART](#) : In this mode two pins will be used. One for normal 8 bit UART data (no parity, one stop bit, LSB first) and one for a framing signal to indicate when a packet begins and ends.
 - [RAIL_PT_MODE_SPI](#) : In SPI mode three pins will be used. One for data, one for the clock, and one for the framing signal.
 - [RAIL_PT_MODE_UART_ONEWIRE](#) : In this mode one pin will be used for 9 bit UART data (no parity, one stop bit, LSB first) where the 9th bit is 1 for control bytes and 0 for normal data. This along with knowledge of the control bytes can be used to parse the data stream.
 - [RAIL_PT_MODE_DISABLED](#) : Turn off the peripheral.
- PTI baudrate
 - The PTI baudrate is created using an 8 bit integer divisor on the radio clock (HFXO) frequency. This limits the selectable range of baud rate values but also means that as the frequency increases the achievable baud rates will be spaced further apart. For this reason SPI mode is preferred at high data rates. Valid ranges for different crystal frequencies are shown below:
 - 38.4 MHz HFXO: Baud rates in the range [149.4kbps, 19.2Mbps]
 - 39.0 MHz HFXO: Baud rates in the range [151.8kbps, 19.5Mbps]

The following hardware options can be changed:

- Configure the DOUT GPIO pin. DOUT is needed for all PTI modes (UART, UART Onewire, SPI) and contains the actual serial data.
- Configure the DFRAME GPIO pin. DFRAME is needed for UART and SPI modes only and is used to frame the start and end of packets. This is not used for UART Onewire mode.
- Configure the DCLK GPIO pin. DCLK is needed for SPI mode only.

Power Amplifier (PA) Utility

Power Amplifier (PA) Utility

RAIL PA Utility

This optional software component can be enabled to include default functionality related to PA configuration.

When using a Silicon Labs-developed board, the configuration options for this software component are set up automatically based on the selected board. When using a custom board, manually configure the configuration options.

Configuration Options

The following configuration options can be changed:

- PA Configuration
 - Initial PA power
 - PA ramp time
 - PA ramp time is a hardware approximation aimed at being as close to the desired ramp time as possible without going over the specified time.
 - Milli-volts provided to the PA supply pin (PA_VDD)
 - 2.4 GHz PA selection (for applicable devices)
 - Sub-1 GHz PA selection (for applicable devices)
- PA Curve Configuration
 - Header file containing custom PA curves
 - Header file containing PA curve type types
- PA Calibration Configuration
 - Apply PA Calibration Factory Offset
 - PA calibration offset ensures that PA power remains consistent chip-to-chip. These adjustments occur automatically when the PA is in use, and the application is not notified when these adjustments occur.

See these documents for in-depth explanations of how to configure and use the available PA features:

- [AN1127: Power Amplifier Power Conversion Functions in RAIL 2.x](#)

Protocol Utility

Protocol Utility

RAIL Protocol Utility

This optional software component can be enabled to include default functionality related to protocol configuration.

This RAIL utility permits the configuration of protocol-specific settings. Each of these protocols can be configured:

- Bluetooth LE
- IEEE 802.15.4, 2.4 GHz
- IEEE 802.16.5, GB868 (sub 1-GHz)
- Z-Wave

Configuration Options

The following configuration options can be changed:

- Timing configurations common to every protocol
 - Transition Times
 - Transition time (microseconds) from idle to RX
 - Transition time (microseconds) from TX to RX
 - Transition time (microseconds) from idle to TX
 - Transition time (microseconds) from RX to TX
 - RX Search Timeouts
 - Enable RX Search timeout after idle
 - Enable RX Search timeout after TX
- IEEE 802.15.4, 2.4 GHz-specific configurations
 - Node Configurations
 - Enable PAN Coordinator
 - Enable Promiscuous Mode
 - Enable default Frame Pending bit value for outgoing ACKs in response to Data Request Command
 - Receivable Frame Types
 - Beacon Frames
 - Data Frames
 - ACK Frames
 - Command Frames
 - Auto ACKs
 - RX ACK timeout duration (microseconds)
 - Radio state transition after attempting to receive ACK
 - Radio state transition after transmitting ACK
- IEEE 802.16.5, GB868 (sub 1-GHz)-specific configurations
 - The same protocol-specific configurations exist for this protocol as exist for IEEE 802.15.4, 2.4 GHz, above.
- Z-Wave-specific configurations
 - Node Configurations
 - Enable Promiscuous Mode
 - Accept Beam Frames
 - Filter Packets Based on Node ID

Recommended Utility

Recommended Utility

RAIL Recommended Utility

This optional software component can be enabled to include a recommended collection of other optional RAIL utilities.

This RAIL utility attempts to identify those RAIL utilities most commonly used by applications. If a list of utilities different from what is auto-selected here is sought, this component must be removed from the project with the custom list manually being added to the project.

Common RAIL utilities recommended for general use include:

- [Initialization](#)
- [Callbacks](#)
- [Protocol](#)
- [Power Amplifier \(PA\)](#)
- [Packet Trace Interface \(PTI\)](#)
- [Received Signal Strength Indicator \(RSSI\)](#)
- [Direct Memory Access \(DMA\)](#)
 - This primarily accelerates radio initialization, so it is currently included only for dynamic multiprotocol (DMP) scenarios.

RF Path Utility

RF Path Utility

RAIL RF Path Utility

This optional software component can be enabled to include default functionality related to device-internal and device-external RF path configuration.

When using a Silicon Labs-developed board, the configuration options for this software component are set up automatically based on the selected board. When using a custom board, manually configure the configuration options.

Note

- This plugin is unnecessary if the project already contains [Antenna Diversity Utility](#).

Configuration Options

The following configuration options can be changed:

- Device-internal (EFR32XG21, EFR32XG23, and EFR32XG28 only)
 - Switch between 2 RF paths (i.e., device pins).

Received Signal Strength Indicator (RSSI) Utility

Received Signal Strength Indicator (RSSI) Utility

RAIL RSSI Utility

This optional software component can be enabled to include default functionality related to RSSI configuration.

To make RSSI values more consistent among families of chips, this software offset can be automatically applied.

Configuration Options

The following configuration options can be changed:

- Software RSSI offset value

RAILtest Application Core

RAILtest Application Core

RAILtest Core

This software component contains the core functionality of RAILtest and must be enabled for RAILtest to operate.

Configuration Options

The following configuration options can be changed:

- Enable/disable external radio configuration file support (e.g., rail_config.c/h).
- Configure default radio configuration for use.
- Alter application name output over CLI on boot. Default is `RAILtest`.
- Alter transfer period between consecutively transmitted packets.
- Alter maximum packet length. This is the size of the TX buffer in the RAILtest application, which contains the packet eventually to be loaded into the RAIL library's TX FIFO for transmission.
- Alter transmit buffer size. This is the size of the TX FIFO provided to the RAIL library.
- Alter receive buffer size. This is the size of the RX FIFO provided to the RAIL library.
- Alter the duration that distinguishes between short and long button presses.

The following hardware options can be changed:

- Configure the Packet Error Rate (PER) GPIO pin. By default, this RAILtest output GPIO pin is not configured for use. For additional information regarding its use in RAILtest, see the section **Packet Error Rate Testing** in the document **UG409: RAILtest User's Guide**.

Various RAILtest-specific buffer dependencies exist. For example, to transmit and receive a 4096 byte packet (i.e., the largest packet size supported in TX and RX packet modes), the buffers must be configured as follows:

- Note: TX and RX buffer sizes provided to the RAIL library must comply with the sizing constraints identified in the section **Receive and Transmit FIFO Buffers** of [EFR32](#).
- TX device
 - `SL_RAIL_TEST_MAX_PACKET_LENGTH = 4096`
 - This is the size of the TX buffer used by the RAILtest application to contain the packet that will eventually be loaded into the library's TX FIFO.
 - Up to 4096 bytes may be transmitted if the receiving device is not collecting the RX packet's appended info (e.g., no CRC status, no RSSI value, no timestamp).
 - Up to 4090 bytes may be transmitted if the receiving device is collecting the RX packet's appended info (e.g., CRC status, RSSI value, timestamp).
 - `SL_RAIL_TEST_TX_BUFFER_SIZE = 4096 // must be power of 2`
 - This is the size of the TX FIFO made available to the RAIL library.
- RX device
 - `BUFFER_POOL_ALLOCATOR_BUFFER_SIZE_MAX = 4144`
 - This is the size of the buffer used by the RAILtest application for CLI output purposes. See the [Memory Manager](#) software component to configure this value.
 - `4144 = receive 4096 byte packet + 0 bytes appended info (i.e., no CRC status, no RSSI value, no timestamp) + 48 bytes extra info for printf (i.e., sizeof(RailEvent_t))`
 - `4144 = receive 4090 byte packet + 6 bytes appended info (i.e., CRC status, RSSI value, timestamp) + 48 bytes extra info for printf (i.e., sizeof(RailEvent_t))`
 - `SL_RAIL_TEST_RX_BUFFER_SIZE = 4096 // must be power of 2`
 - This is the size of the RX FIFO made available to the RAIL library.

RAILtest Application Graphics

RAILtest Application Graphics

RAILtest Graphics

This software component can optionally be enabled for RAILtest. If enabled, the following information is displayed on the LCD:

- Number of transmitted packets
- Number of received packets
- Channel number
- Application name

When using a Silicon Labs-developed board, the hardware GPIO configuration is set up automatically for this software component based on the selected board. When using a custom board, manually configure the hardware pins. This can be done from within the `Memory LCD` software component. The `Memory LCD` component determines which LCD displays are compatible with the `RAILtest, Graphics` component. (The `Memory LCD` component is a dependency of the `RAILtest, Graphics` component and is included automatically in a project when the `RAILtest, Graphics` component is enabled.)

Configuration Options

The following configuration options can be changed:

- Alter application name displayed on boot. Default is `RAILtest`.

Thermistor Utility

Thermistor Utility

RAIL Thermistor Utility

This optional software component can be enabled to include advanced functionalities related to temperature on target having [RAIL_SUPPORTS_EXTERNAL_THERMISTOR](#) bit set.

The thermistor component allows the user to enabled multiple features related to temperature such as:

- High Frequency Crystal Oscillator (HFXO) compensation
- Thermal protection

Note that thermistor can be used independantly of this component using already exisiting API such as:

- [RAIL_StartThermistorMeasurement\(\)](#).
- [RAIL_GetThermistorImpedance\(\)](#).

This component offers a default configuration and conversion functions.

When using a Silicon Labs-developed board with a thermistor, the configuration options for this software component are set up automatically and the conversion functions are based on the selected board. When using a custom board, user must manually configure the configuration options. If the thermistor is custom as well, user must manually override the conversion functions.

Configuration Options

The following hardware options can be changed:

- Configure the THERMISTOR GPIO pin.
- Configure the THERMISTOR GPIO port.

The following conversion functions can be changed:

- Impedance to temperature conversion ([RAIL_ConvertThermistorImpedance\(\)](#)).
- Temperature to PPM deviation conversion ([RAIL_ComputeHFXOPPMError\(\)](#)).

RF Path Switch Utility

RF Path Switch Utility

RAIL RF Path Switch Utility

This optional software component can be enabled to control a GPIO toggled switch based on RFPATH selection and radio state. An example usage of this component is toggling a switch on RFPATH0 to ground when RFPATH1 is in use to avoid spurious harmonics.

When using a Silicon Labs-developed board, the configuration options for this software component are set up automatically based on the selected board. When using a custom board, manually configure the configuration options.

Note

- Usage of this component is only relevant on the dual-band EFR32XG28 where externally grounding the unused RF path prevents spurious harmonics.

Configuration Options

The following configuration macros are available for modification:

- RF Path Switch:
 - SL_RAIL_UTIL_RF_PATH_SWITCH_CONTROL_PORT: GPIO port for rfpath selection signal output.
 - SL_RAIL_UTIL_RF_PATH_SWITCH_CONTROL_PIN: GPIO pin for rfpath selection signal output.
 - SL_RAIL_UTIL_RF_PATH_SWITCH_INVERTED_CONTROL_PORT: GPIO port for logical NOT rfpath selection signal output.
 - SL_RAIL_UTIL_RF_PATH_SWITCH_INVERTED_CONTROL_PIN: GPIO pin for logical NOT rfpath selection signal output.
 - SL_RAIL_UTIL_RF_PATH_SWITCH_RADIO_ACTIVE_PORT: GPIO port for radio active signal output.
 - SL_RAIL_UTIL_RF_PATH_SWITCH_RADIO_ACTIVE_PIN: GPIO pin for radio active signal output.
 - SL_RAIL_UTIL_RF_PATH_SWITCH_RADIO_ACTIVE_MODE: Logical AND radio active signal output with CONTROL or INVERTED_CONTROL signal outputs if set to SL_RAIL_UTIL_RF_PATH_SWITCH_RADIO_ACTIVE_COMBINE.

Radio Sequencer Image Selection Utility

Radio Sequencer Image Selection Utility

RAIL Utility for Selecting a Radio Sequencer Image

This optional software component allows for the selection of a specific radio sequencer image on platforms that support multiple images. The selection can be done automatically based on properties of the OPN, or manually based on the application needs of the user.

Currently, only EFR32XG24 supports multiple sequencer images. The two images available for that platform are for the 10 dBm PA and 20 dBm PA variants. This component will select the correct image automatically based on the OPN. In the future, other sequencer images may be introduced on other platforms that allow the user to choose an application specific custom sequencer image.

On platforms that support only one default sequencer image, this component is safe to include, and will have no effect.

Configuration Options

The following configuration options can be changed:

- Enable/disable run-time selection of the sequencer image.

Software Modem (SFM) sequencer image selection Utility

Software Modem (SFM) sequencer image selection Utility

RAIL Utility for selecting SFM sequencer image

This optional software component allows for the selection of a specific SFM sequencer image on EFR32XG25 platform. Currently, available images are intended for the support of following sets of modulations:

- SUN OFDM + SUN OQPSK
- SUN OFDM
- NONE

EFR32

EFR32

While RAIL attempts to be chip-agnostic, certain hardware specifics can't be overlooked. Where possible, missing features will be simulated in software, but performance characteristics may vary. This section covers EFR32-specific content including hardware-specific configurations or calibrations.

Clocks

High-Frequency Clocks

The clock tree driving the radio subsystem (see the chip specific reference manual) must be sourced from the HFXO or an HFXO-referenced clock source. The radio clock frequency must be either the HFXO's or twice the HFXO's. This generally means that you must switch your HFCLK source to the HFXO before calling [RAIL_Init\(\)](#) and make sure to leave the CMU->HFPRESC set to 1 to prevent dividing down the radio clock. You are free to scale the peripheral (CMU->HFPERPRESC) or Cortex (CMU->HFCOREPRESC) clocks using their prescalars, which are further down the clock tree. Keep in mind that this may introduce extra wait states when interacting with the radio and slow down some operations.

RAIL's built-in radio configuration PHYs for BLE, IEEE 802.15.4, and other protocols generally require the HFXO frequency to be 38.4 MHz except for EFR32xG25 and EFR32xG28 which require 39.0 MHz. A few platforms support a variety of HFXO frequencies for their built-in PHYs, selectable via the [Built-in PHYs for alternate HFXO frequencies](#) component. If using only proprietary radio configuration PHYs, RAIL does not require a particular HFXO frequency but it's strongly suggested to be within the 38.4 MHz to 40 MHz range, and of course that frequency will need to be made known to the radio calculator when generating such proprietary PHYs.

If using the DPLL on EFR32 Series-2 platforms, it is best to configure `cmuDPLLLockMode_Phase` where the frequency error will tend to 0. Using `cmuDPLLLockMode_Freq` should be avoided because it can cause issues with applications requiring very precise frequencies like proprietary protocols or Bluetooth. On EFR32xG22 going to EM1P sleep mode is precluded when an 80MHz HRFCO PLL system clock is selected because it can cause a noticeable clock drift which impacts radio timing and tuning.

Low-Frequency Clocks

RAIL can use a low-frequency clock on the EFR32 to synchronize its time base across EM2 sleep. This only happens if you call [RAIL_ConfigSleep](#) with [RAIL_SleepConfig_t::RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED](#). The slower the clock source, the longer it will take to synchronize because the process requires several ticks of the RTCC clock. When calling [RAIL_ConfigSleep\(\)](#), the code will first try to set up the timer with the LFXO if it's started, then fall back to the LFRCO, and finally assert if no LF clocks are running.

Receive and Transmit FIFO Buffers

The RAIL library for the EFR32 optionally allocates an internal buffer to store receive data contiguously. Its required alignment is 32-bit (4-byte); applications can use [RAIL_FIFO_ALIGNMENT](#) or the [RAIL_FIFO_ALIGNMENT_TYPE](#) to ensure conformance.

The buffer is set to 512 bytes at build time by default, but can be changed by calling [RAIL_SetRxFifo](#), for example in an implementation of [RAILCb_SetupRxFifo](#). The receive FIFO's size must be a power of 2 between 64 and 4096 bytes for hardware compatibility.

The chosen buffer size limits the maximum size of receive packets in packet mode and determines the size of the receive FIFO in FIFO mode. Because each receive packet has several bytes of overhead, you can only receive up to one (buffer size - overhead) byte packet without switching to FIFO mode. In FIFO mode, you must read out packet data as you approach this limit and store it off to construct the full packet later. This overhead is currently 8 bytes on all EFR32 Series-2

platforms. Note that this overhead may increase or decrease in future releases as the functionality is changed though large jumps are not expected in either direction.

Supplementing the receive FIFO used for packet data, the RAIL library also allocates an internal packet metadata FIFO capable of holding up to 16 packets' worth of metadata. An entry in this FIFO is made on every packet completion (successful or not) signaled to the application. If the application calls [RAIL_HoldRxPacket\(\)](#) that entry is held along with any of its packet data in the receive FIFO. When either the receive FIFO or internal packet metadata FIFO fills or overflows, [RAIL_EVENT_RX_FIFO_FULL](#) or [RAIL_EVENT_RX_FIFO_OVERFLOW](#) occur at packet completion time, respectively.

The transmit FIFO must be set at runtime with [RAIL_SetTxFifo\(\)](#). Like the receive FIFO the transmit FIFO's required alignment is 32-bit (4-byte); applications can use [RAIL_FIFO_ALIGNMENT](#) or the [RAIL_FIFO_ALIGNMENT_TYPE](#) to ensure conformance.

The transmit FIFO's size has the same restrictions as the receive FIFO. The size must be a power of 2 from 64 to 4096 for hardware compatibility. Note that there is no difference between packet and FIFO mode on the transmit side. A packet may either be loaded all at once or in pieces using the [RAIL_WriteTxFifo\(\)](#) function. You may also use the [RAIL_SetTxFifoThreshold\(\)](#) function and the [RAIL_EVENT_TX_FIFO_ALMOST_EMPTY](#) event to load data as there is space available in the FIFO during transmission.

RAIL also provides an internal transmit AutoAck FIFO whose size is fixed at [RAIL_AUTOACK_MAX_LENGTH](#) bytes. Applications can write to it via [RAIL_WriteAutoAckFifo\(\)](#). Since RAIL does not permit the queuing of multiple transmits, there is no need for any internal transmit metadata FIFO like the one which exists on the receive side.

Data Reception Sources

When receiving data, you can configure hardware to provide data from three different hardware sources.

First, you can configure the hardware to provide a packet of information. This configuration uses the built-in demodulator and frame controller. Use [RAIL_RxDataSource_t::RX_PACKET_DATA](#) to enable this receive source.

Second, you can configure hardware to provide data directly from the demodulator. In this mode, hardware demodulation is used, but the user is responsible for implementing frame controller functionality. In other words, preamble detection, sync word detection, CRC validation, and so on must be performed in software by the application. All data returned is represented as 8-bit 2's-complement values. Use [RAIL_RxDataSource_t::RX_DEMOD_DATA](#) to enable the receive source.

Third, you can configure hardware to provide data directly from the I and Q ADCs. In this mode, the user is responsible for implementing demodulator and frame controller functionality. The receive signal hardware has a 19-bit dynamic range. The user can select whether to return the upper 16 bits of the 19-bit value ([RAIL_RxDataSource_t::RX_IQDATA_FILTMSB](#)) or the lower 16 bits ([RAIL_RxDataSource_t::RX_IQDATA_FILTLSB](#)). All data returned is represented as 16-bit 2s-complement values. The I and Q values are put into the buffer in an alternating fashion where the first 16-bit value is the first I ADC value and the second 16-bit value is the first Q ADC value, and so on.

Interrupt Vectors

The RAIL library for EFR32 implements all of the interrupt vectors for radio peripherals, which is required for the RAIL library to function correctly. RAIL does not, however, set the priorities of these interrupt vectors. The others must be handled by your application using the CMSIS NVIC_SetPriority() API or direct manipulation of the NVIC. Below is the full list of interrupts used by the radio. You **must** run them all except FRC_PRI_IRQHandler at the same priority which is what all Silicon Labs RAIL applications do by default. You are free to choose what that priority is based on the requirements of your application (e.g., putting them below FreeRTOS OS atomic for access to OS functions in RAIL callbacks). Keep in mind that putting the radio interrupts at too low a priority can cause missed packets and other radio performance issues. A restriction on EFR32 Series-2 is that you must not disable radio interrupts for longer than a quarter of the RAIL timebase ($2^{32}/4$ microseconds or 18 minutes) to ensure proper timekeeping. See [RAIL_EFR32xG2x_Interrupts](#) for more information.

Chip-Specific Initialization

EFR32 Hardware Initialization

EFR32 includes EMLIB and EMDRV to create a basic HAL layer. A lot of this initialization code is completely up to you, but there are a couple of requirements when building a RAIL app. Specifically, the radio will only work if you are running off a

high-precision crystal oscillator. Since some APIs will assume this is running, make sure to initialize and switch to the crystal before calling any radio APIs.

For the Wireless Starter Kit (WSTK), you can use the crystal configuration in the HAL configuration header file for your specific kit. Example code for this is shown below. If you have a custom hardware layout, you may want to create your own HFXOInit structure to account for things such as your specific CTUNE value.

```
#include "bsp.h" // Contains WSTK versions of the HFXO init structure

void efrStartup(void)
{
    CMU_HFXOInit_TypeDef hfxoInit = CMU_HFXOINIT_WSTK_DEFAULT;

    // Initialize the HFXO using the settings from the WSTK bspconfig.h
    // Note: This configures things like the capacitive tuning CTUNE variable
    // which can vary based on your hardware design.
    CMU_HFXOInit(&hfxoInit);

    // Switch HFCLK to HFXO and disable HFRCO
    CMU_ClockSelectSet(cmuClock_HF, cmuSelect_HFXO);
    CMU_OscillatorEnable(cmuOsc_HFRCO, false, false);
}
```

Radio-Specific Initialization

Currently, [RAIL_ConfigTxPower\(\)](#) and [RAIL_ConfigPti\(\)](#) initialization functions depend on the board configuration and must be manually called at startup. You **MUST** initialize the power amplifier (PA) to transmit. You may initialize the packet trace interface (PTI) for use in debugging.

Power Amplifier (PA) Initialization

The specific set of PAs you have available and the supported power levels for those PAs are determined by your part number. See the appropriate data sheet for more details.

Each PA supports a raw power level value of type [RAIL_TxPowerLevel_t](#) where a numerically smaller number causes less power to be output from the chip than a higher number. Keep in mind that these values may be capped at the upper and lower ends and do not necessarily map linearly to output power in dBm. To map these values to and from dBm values, the RAIL library uses the [RAIL_ConvertRawToDbm\(\)](#) and [RAIL_ConvertDbmToRaw\(\)](#) APIs. By default, these use whatever piecewise linear power curves were passed into [RAIL_InitTxPowerCurves\(\)](#) to do the conversion. On Silicon Labs boards, these curves are included with the board header files. For custom boards, this must be measured using test equipment to take into account the differences in the RF path that may exist. In addition, for even more custom use cases, the conversion functions can be strongly defined in the customer application and provide whatever type of mapping is found to be most effective. Below is an example of using the standard APIs to initialize all PAs on a dual band chip and switch to the 2.4 GHz high-power PA.

```
#include "rail.h"
#include "rail_chip_specific.h"
#include "plugin/pa-conversion/pa_conversion_efr32.h"

RAIL_DECLARE_TX_POWER_VBAT_CURVES(piecewiseSegments, curvesSg, curves24Hp, curves24Lp);

// Must be called with a valid RAIL_Handle_t returned by RAIL_Init()
void initPa(RAIL_Handle_t myRailHandle)
{
    // Initialize the RAIL Tx power curves for all PAs on this chip
    RAIL_TxPowerCurvesConfig_t txPowerCurvesConfig = {
        curves24Hp,
        curvesSg,
        curves24Lp,
        piecewiseSegments
    };
    if (RAIL_InitTxPowerCurves(&txPowerCurvesConfig) != RAIL_STATUS_NO_ERROR) {
        // Could not initialize transmit power curves so something is configured
        // wrong. Please fix and rebuild.
        while(1);
    }

    // Switch to the 2.4GHz HP PA powered off the 1.8V DCDC connection
    RAIL_TxPowerConfig_t railTxPowerConfig = {
        RAIL_TX_POWER_MODE_2P4GIG_HP, // 2.4GHz HP Power Amplifier mode
        1800, // 1.8V vPA voltage for DCDC connection
        10 // Desired ramp time in us
    };
    if (RAIL_ConfigTxPower(myRailHandle, &railTxPowerConfig)
        != RAIL_STATUS_NO_ERROR) {
        // Error: The requested PA could not be selected. Fix your configuration
        // and try again.
        while(1);
    }

    // Set the output power to the maximum supported by this chip
    RAIL_SetTxPower(myRailHandle, 255);
}
```

Packet Trace Interface (PTI) Initialization

Packet trace on the EFR32 provides a mechanism for viewing transmitted and received radio packets for network sniffing or debugging. It can also be captured by a WSTK and sent to Simplicity Studio for viewing data in its Network Analyzer tool.

To use this functionality, configure the pins to use for this output and, optionally, how to format data. Note that the WSTK **requires** the following output format:

```
Mode: 8 bit UART mode
Baudrate: 1.6 Mbps
Framing Signal: Enabled
```

Introducing changes is currently unsupported by the WSTK. Other output formats are currently unsupported.

To choose the pins, look in the applicable data sheet and select a valid route location for the FRC_DOUT and FRC_DFRAME signals. These map to PTI.DATA and PTI.FRAME on the WSTK respectively. Once you've found pins that work for your hardware, configure the RouteLocation, port, and pin fields in the PTI initialization structure. When using the WSTK for example, initialize as follows:


```
#include "rail.h"
#include "sl_gpio.h"

void initPti(void)
{
    RAIL_PtiConfig_t ptiConfig = {
        .mode = RAIL_PTI_MODE_UART, // Only supported output mode for the WSTK
        .baud = 1600000, // Choose 1.6 Mbps for the WSTK
        .doutPort = SL_GPIO_PORT_B, // PB12
        .doutPin = 12, // PB12
        // .dclkPort = not unused in UART mode,
        // .dclkPin = not unused in UART mode,
        .dframePort = SL_GPIO_PORT_B, // PB13
        .dframePin = 13, // PB13
    };

    // Initialize the Packet Trace Interface (PTI) for the EFR32
    RAIL_ConfigPti(RAIL_EFR32_HANDLE, &ptiConfig);

    // Enable Packet Trace (PTI)
    RAIL_EnablePti(RAIL_EFR32_HANDLE, true);
}
```

Other Radio GPIO Functions

Various useful signals related to the radio can be output on a GPIO using the Peripheral Reflex System (PRS). PRS is an advanced system where you can route signals to channels and then output those channels to a number of configurable locations. For more information, see the PRS chapter in the reference manual.

A list of some of the most interesting PRS signals related to the radio is shown below along with how to enable them in the PRS. The definition for these signals for a given chip can be found in the release at `platform/Device/SiliconLabs/<chipFamily>/Include/<chipFamily>_prs_signals.h` where `<chipFamily>` would be the beginning of the part number (EFR32MG21 for example).

Signal	Summary
PRS_RAC_ACTIVE	Radio enabled
PRS_RAC_TX	Transmit mode enabled
PRS_RAC_RX	Receive mode enabled
PRS_RAC_LNAEN	LNA enabled for RX
PRS_RAC_PAEN	PA enabled for TX
PRS_MODEM_FRAMEDET	Frame detected
PRS_MODEM_PREDET	Preamble detected
PRS_MODEM_TIMDET	Timing detected
PRS_MODEM_FRAMESENT	Frame sent
PRS_MODEM_SYNCSENT	Sync word sent
PRS_MODEM_PRESENT	Preamble sent

The example below shows how to configure a PRS channel to output RAC_RX on a GPIO. In this example, the WSTK is used with the BRD4153A radio board. The code below will put RAC_RX on PRS Channel 0 and output PRS Channel 0 on pin PC10 which is wired to WSTK_P12 and EXP_HEADER15 on the WSTK.

```
#include "em_device.h"
#include "sl_gpio.h"
#include "em_cmu.h"
#include "em_prs.h"

void enableDebugGpios(void)
```

```
// Turn on the PRS and GPIO clocks to access their registers.CMU_ClockEnable(cmuClock_PRS,true);CMU_ClockEnable(cmuClock_GPIO,true);//
Configure PC10 as an output.sl_gpio_set_pin_mode(PC10, SL_GPIO_MODE_PUSH_PULL,false);// Configure PRS Channel 0 to output
RAC_RX.PRS_SourceSignalSet(0,((PRS_RAC_RX & _PRS_CH_CTRL_SOURCESEL_MASK)>> _PRS_CH_CTRL_SOURCESEL_SHIFT),((PRS_RAC_RX &
_PRS_CH_CTRL_SIGSEL_MASK)>> _PRS_CH_CTRL_SIGSEL_SHIFT),
    prsEdgeOff);// Configure PRS Channel 0 to use output location 12 (PC10 - see data sheet).
PRS->ROUTELOC0 &= ~PRS_ROUTELOC0_CH0LOC_MASK;
PRS->ROUTELOC0 |= PRS_ROUTELOC0_CH0LOC_LOC12;// Enable PRS Channel 0.
PRS->ROUTEPEN |= PRS_ROUTEPEN_CH0PEN;}
```

Required Dependencies

Most of the RAIL library is self-contained, however, there are some dependencies on external functions, such as functions from the C standard library, EMLIB, and CMSIS. Below is a complete list of these dependencies. Note that changing the implementation of any of these functions while maintaining their functionality could still impact RAIL operation.

Group	Functions
CMSIS	SystemHFRCODPLLClockGet() SystemHFXOClockGet() SystemLFRCOClockGet() SystemLFXOClockGet() SystemSYSCLKGet() SystemULFRCOClockGet()
DRIVER (sl_gpio)	sl_gpio_set_pin_mode()
EMLIB (em_cmu)	CMU_ClockEnable() CMU_ClockFreqGet() CMU_ClockSelectGet() CMU_HFXOCTuneDeltaGet() CMU_HFXOCTuneDeltaSet() CMU_HFXOCTuneGet() CMU_HFXOCTuneSet() CMU_HFXOCoreBiasCurrentCalibrate()
EMLIB (em_core)	CORE_EnterCritical() CORE_ExitCritical() CORE_EnterAtomic() CORE_ExitAtomic()
EMLIB (em_emu)	EMU_DCDCSetPFMXTimeoutMaxCtrl()
EMLIB (em_idma)	LDMA_StartTransfer() LSMA_StopTransfer() LDMA_TransferDone()
EMLIB (em_system)	SYSTEM_ChipRevisionGet()
stdlib	mcmcmp() memcpy() memset()

Peripherals Consumed by RAIL

Certain functionality in RAIL requires some of the chip's peripherals. The specifics of these requirements are enumerated below.

TrustZone Considerations

RAIL internally accesses CMU, EMU, GPIO, HFXO and LDMA as noted above, along with its own radio-related peripherals, using TrustZone non-secure addresses. On TrustZone platforms (all Series 2) Trustzone must be initialized/enabled and these peripherals configured non-secure. RAIL does not support trustzone_unaware applications. Furthermore these peripherals' configurations should not be locked by the peripherals' corresponding LOCK register.

Certain platforms where [RAIL_SUPPORTS_TRUSTZONE_SECURE_PERIPHERALS](#) is true may allow an application to use RAIL when some of these non-radio peripherals are configured secure. In this case some RAIL code must be executed as secure and some RAIL functionality is necessarily limited depending on which peripherals are made secure. Refer to [TrustZone](#) for more details.

RAIL Timer Synchronization

If you call [RAIL_ConfigSleep](#) with [RAIL_SleepConfig_t::RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED](#) to keep the clock synchronized across sleep RAIL must use the PRS and RTCC.

- On all platforms, PRS channel 7 is used to perform the synchronization. This channel is only initialized one time when calling the [RAIL_ConfigSleep](#) function so you must not change it after that or the synchronization may fail.
- On the EFR32xG21 platform, the timer synchronization also uses RTCC channel 0. This channel should not be used by the application after a call to [RAIL_Sleep](#) and before the corresponding [RAIL_Wake](#) call. The application is also responsible for enabling this top level interrupt if they want it to serve as a wake source and implementing an interrupt handler that clears the RTCC_IFC_CC0 flag in the RTCC->IF register any time it is pending to prevent the chip from getting stuck in the handler. RAIL does not need to be tied into the interrupt handler in any other way.

DCDC

If you enable or disable the DCDC peripheral, you must follow it by calling [RAIL_ChangedDcdc](#). This API indicates that the DCDC peripheral bus clock has changed allowing RAIL to react accordingly.

LDMA

RAIL's use of LDMA is generally optional and enabled only by calling [RAIL_UseDma\(\)](#). It can be speed up [RAIL_Init\(\)](#) initialization. Some RAIL features, like [RAIL_IEEE802154_ConfigRxChannelSwitching\(\)](#) and related [RAIL_RX_OPTION_CHANNEL_SWITCHING](#), may require the use of LDMA and won't function without it being configured.

Entropy Generation

EFR32 supports true entropy collection using the radio. It is able to collect 1 bit per radio clock cycle while in receive mode. This means that the receiver must be enabled when collecting entropy. If you attempt to transmit or otherwise delay entry into receive, data collection will take longer. You can still receive packets while collecting entropy. This means that requesting random numbers longer than the length of your preamble and sync word may trigger a packet reception. This may be fixed in a future revision but for now it ensures a proper packet reception during a random data collection.

Due to the nature of random data collection, either the number of bytes or zero is returned after the full amount requested. Zero is returned if the radio is uninitialized and cannot be enabled for collection.

RAIL Timebase

EFR32 uses a dedicated radio timer to create the RAIL timebase and perform scheduled transmit and receive operations. This timer ticks at roughly every 0.5 us on EFR32 Series-2. The exact tick value depends on the high-frequency crystal in use so there may be a small rounding error. All internal APIs account for this error, so over long periods of time it is generally unnoticeable. Note that RAIL uses the hardware tick rate to create a 1 us time base that is exposed to the user. This is all that should matter for most users and the information here is included to explain the underlying implementation.

Radio Calibration

EFR32 supports image rejection (IR) calibrations, VCO temperature calibration, and HFXO compensation. You may choose to enable some or all depending on your use case. Using IRCAL and VCO_TEMPCAL is recommended for optimal performance of the radio.

Image Rejection Calibration (IRCAL)

This calibration should be run each time your PHY configuration changes. It is based on the modulation scheme, frequency band, and other radio settings. RAIL will request this by calling [RAIL_Config_t::eventsCallback](#) with the [RAIL_EVENT_CAL_NEEDED](#) bit set and the [RAIL_CAL_ONETIME_IRCAL](#) bit set in [RAIL_GetPendingCal\(\)](#).

Using a proper value for this will improve sensitivity by several dBm, so it's highly recommended.

It can take on the order of 700 ms to complete this calibration. You may want to save off a known good value for this calibration and load it each time you switch PHYs to save time.

This calibration should be initialized before calling [RAIL_ConfigChannels\(\)](#) to ensure that it is properly configured by the time the first channel is set up. The initialization involves enabling the algorithm and passing in parameters to configure the algorithm for the specific PHY generated by the EFR32 Radio Configurator.

You cannot use the radio while this calibration is being performed or you may generate an incorrect calibration. Application code should ensure that the radio remains in the idle state during this calibration.

This calibration is only meaningful for Zigbee, BLE, and sub GHz radio configurations. For all others you should still enable the calibration but the algorithm will apply a safe, default, calibration value.

Image Rejection Calibration for Transmit (TXIRCAL)

This calibration is only meaningful for sub GHz radio configurations using SUN-OFDM PHYs and it should be run only once per multi-PHY configuration. For all other configurations this calibration won't run. RAIL will request this by calling [RAIL_Config_t::eventsCallback](#) with the [RAIL_EVENT_CAL_NEEDED](#) bit set and the [RAIL_CAL_ONETIME_IRCAL](#) bit set in [RAIL_GetPendingCal\(\)](#).

This feature is automatically run after IRCAL, when a SUN-OFDM PHY is configured. If a SUN-OFDM PHY exists in the multi-PHYs radio configuration, it is advised to first load this OFDM PHY and then call [RAIL_Calibrate\(\)](#), so that both the IRCAL and TXIRCAL are run immediately. Otherwise the calibration will be requested the first time an OFDM PHY will be loaded.

Like IRCAL, using a proper value for this will improve sensitivity by several dBm, so it's highly recommended.

It can take on the order of 150 ms to complete this calibration. You may want to save off a known good value for this calibration and load it each time you switch multi-PHY configuration to save time.

You cannot use the radio while this calibration is being performed or you may generate an incorrect calibration. Application code should ensure that the radio remains in the idle state during this calibration.

VCO Temperature Calibration (VCO_TEMPCAL)

When staying in receive for a very long time and the temperature changes causing the radio to drift off-frequency, RAIL will request this calibration via [RAIL_Config_t::eventsCallback](#) with the [RAIL_EVENT_CAL_NEEDED](#) bit set and the [RAIL_CAL_TEMP_VCO](#) bit set in [RAIL_CalPendingGet\(\)](#).

This calibration is automatically run every time receive is entered. If the application, by its nature, frequently re-enters receive mode, this calibration may not need to be enabled.

On EFR, the application will get this event when the absolute temperature crosses 0C degrees as well as when the temperature delta from the last calibration increases or decreases by 70C.

It is always recommended to enable and handle this calibration since it doesn't add much overhead and is much safer.

HFXO Temperature Compensation (HFXO_TEMPComp)

This optional feature requires plugin [Thermistor Utility](#). More details about the compensation are available in the plugin page.

When the temperature varies too much, HFXO frequency drifts, resulting in signal degradation. The compensation corrects the effects of this deviation in the radio, and is done in 2 steps (2 events).

The first event is [RAIL_EVENT_CAL_NEEDED](#) with the [RAIL_CAL_TEMP_HFXO](#) bit set in [RAIL_GetPendingCal\(\)](#). This signals the need to call [RAIL_StartThermistorMeasurement\(\)](#).

The second event occurs at the end of this measurement triggering [RAIL_EVENT_THERMISTOR_DONE](#) with the [RAIL_CAL_COMPENSATE_HFXO](#) bit set in [RAIL_GetPendingCal\(\)](#). This signals the need to idle the radio and call [RAIL_CalibrateHFXO\(\)](#).

When enabled, [RAIL_EVENT_CAL_NEEDED](#) event is automatically set when the number of degrees set in either [RAIL_HFXOCompensationConfig_t::deltaNominal](#) or [RAIL_HFXOCompensationConfig_t::deltaCritical](#) are exceeded.

State Transition Timing

EFR32 allows automatic transitions from receive to transmit to precisely time transmitted packets after packet reception. During this process, internal calculations are performed which require the received packet's duration, from sync word to end of CRC, to be less than 32 ms. If received packets in a given protocol have an on-air time greater than 32 ms, the automatic transition from receive to transmit may not be used with that protocol because of incorrect timing of that transition.

Radio State Verification

RAIL includes a radio state verification feature capable of verifying radio register contents. There are multiple ways of configuring and running the verification process. All radio state verification should occur when the radio is idle. If not idle, the number of radio registers different from their reference values varies at any given time.

Default vs. Custom Radio Configuration

When verification occurs, a reference value is compared with a radio register's contents. The reference value is provided through one of two ways.

The radio configuration (originally passed into [RAIL_ConfigChannels\(\)](#)) is used as a reference with which to compare radio register values. When the default radio configuration is used for verification, only those addresses flagged as being verifiable by the radio configurator will be checked. Bit 27 of an encoded address indicates whether that address is verifiable or not. For example, encoded address 0x08020004 is flagged as being verifiable and will be verified when [RAIL_Verify\(\)](#) is called, but the encoded address 0x00020008 will not be verified.

A custom radio configuration can be provided to the verification API (passed into [RAIL_ConfigVerification\(\)](#)). When a custom radio configuration is provided, all addresses in the provided radio configuration will be checked, regardless of whether or not their encoded addresses are flagged as being verifiable.

Approval Callback

When verification occurs, no register value differences are allowed unless an application-level approval callback is defined.

Without an approval callback, any difference will be flagged as data corruption, with [RAIL_Verify\(\)](#) returning a status of [RAIL_STATUS_INVALID_STATE](#).

With an approval callback, the application can scrutinize all differences and deem them acceptable or not. By providing an approval callback, individual register fields, instead of the entire register contents, can be compared against reference values at the application level.

Restarting

Verification can be restarted if necessary.

When running verification from the beginning to completion (by specifying a sufficiently long test duration or by specifying a duration of [RAIL_VERIFY_DURATION_MAX](#)), the verification process is indifferent to the restart parameter.

When running verification for a duration that does not permit verification to run to completion, the [RAIL_Verify\(\)](#) function will return with a status of [RAIL_STATUS_SUSPENDED](#). This indicates test success for those values already verified, but sufficient time was not permitted to verify all values. In this scenario, the initial run of [RAIL_Verify\(\)](#) is indifferent to the value of the restart input, but subsequent runs of [RAIL_Verify\(\)](#) should set the restart input to false so that verification will resume where it left off on the previous call. To record where a previous verification left off, the [RAIL_VerifyConfig_t](#)

structure must be declared by the application and provided to RAIL. These structure contents should only be altered by RAIL.

Configuration and Running Options

With the ability to specify default radio configuration vs. custom radio configuration, approval callback vs. no approval callback, and run to completion vs. run for a specific time, there are multiple ways in which to run RAIL's radio state verification feature. The example below shows running verification in each of these modes.

```

#include "rail.h"
#include "rail_types.h"
#include "rail_config.h"

RAIL_VerifyConfig_t configVerify;

const uint32_t customRadioConfig[] = {
    0x00010048UL, 0x00000000UL,
    0x000400A0UL, 0x00004CFFUL,
    /* 00A4 */ 0x00000000UL,
    /* 00A8 */ 0x00004DFFUL,
    /* 00AC */ 0x00000000UL,
    0x00012000UL, 0x00000744UL,
    ...
    0xFFFFFFFFUL,
};

bool RAILCb_VerificationApproval(uint32_t address,
                                uint32_t expectedValue,
                                uint32_t actualValue)
{
    responsePrint("verifyRadioCb",
                  "address:0x%08x,expectedValue:0x%08x,actualValue:0x%08x",
                  address, expectedValue, actualValue);
    return true OR false; // true = change approved; false = change unapproved
}

main ()
{
    RAIL_Status_t result;
    ...
    // Initialize the radio.
    railHandle = RAIL_Init(..., ...);
    ...
    // Associate a radio config with this RAIL handle.
    RAIL_ConfigChannels(railHandle, channelConfigs[0], ...);
    ...
    // Idle the radio before verifying registers.
    RAIL_Idle(railHandle, RAIL_IDLE, true);

    uint8_t verifyConfigOption = 1; // 1,2,3,4
    uint8_t verifyRunOption    = 1; // 1,2

    // Configure the radio state verification in one of 4 ways:
    if (1 == verifyConfigOption) {
        // Config Option 1: Use default radio config with no approval callback.
        RAIL_ConfigVerification(railHandle,
                               &configVerify,
                               NULL, // use channelConfigs[0] for verification
                               NULL);
    } else if (2 == verifyConfigOption) {
        // Config Option 2: Use default radio config with approval callback.
        RAIL_ConfigVerification(railHandle,
                               &configVerify,
                               NULL, // use channelConfigs[0] for verification
                               RAILCb_VerificationApproval);
    } else if (3 == verifyConfigOption) {
        // Config Option 3: Use custom radio config with no approval callback.
        RAIL_ConfigVerification(railHandle,
                               &configVerify,
                               &customRadioConfig[0],
                               NULL);
    } else {
        // Config Option 4: Use custom radio config with approval callback.
        RAIL_ConfigVerification(railHandle,
                               &configVerify,
                               &customRadioConfig[0],

```

```

        RAILCb_VerificationApproval);}elseif(3== verifyConfigOption){// Config Option 3: Use custom radio config with no approval
callback.RAIL_ConfigVerification(railHandle,&configVerify,&customRadioConfig[0],
        NULL);}else{// Config Option 4: Use custom radio config with approval
callback.RAIL_ConfigVerification(railHandle,&configVerify,&customRadioConfig[0],
        RAILCb_VerificationApproval);}// Run the radio state verification in one of 2 ways:if(1== verifyRunOption){// Run Option 1: Run
verification once to completion.
    result =RAIL_Verify(railHandle,&configVerify,
        RAIL_VERIFY_DURATION_MAX,true);// run from the beginning of verification}else{// Run Option 2: Run verification for a specific time
before returning.
    result =RAIL_Verify(railHandle,&configVerify,100,// run for 100 microseconds before returningtrue);// run from the beginning of
verificationwhile(RAIL_STATUS_SUSPENDED == result){responsePrint("verifyRadio","verification suspended - run again to continue");
    result =RAIL_Verify(railHandle,&configVerify,100,// run for 100 us before returningfalse);// run from where verification left off}}

switch (result){
case RAIL_STATUS_NO_ERROR:{responsePrint("verifyRadio","success, done");break;}
case RAIL_STATUS_INVALID_PARAMETER:{responsePrint("verifyRadio","invalid input parameter");break;}
case RAIL_STATUS_INVALID_STATE:
default:{responsePrint("verifyRadio","data corruption");}}...}

```

RFSENSE

The RAIL library for EFR32 supports RF Sensing on EFR32xG22 only. Enabling RFSENSE will enable the ULFRCO clock, if not already enabled.

Energy Detection Mode

This RFSENSE mode detects the presence of RF energy and triggers an event if that energy is continuously present for certain duration of time. The RFSENSE energy duration ranges from 1 ms - 128 ms as a power of 2. Based on the user's RFSENSE duration (senseTime) input, the duration is checked against the actual period and the next higher period. If the senseTime is greater than or equal to $((\text{actualPeriod} + \text{nextHigherPeriod}) / 2)$, the next higher duration is chosen. The energy detection threshold can be configured for high sensitivity (low noise environment) or low sensitivity (high noise environment) by selecting appropriate RF Band via [RAIL_RfSenseBand_t](#).

Selective Mode

This RFSENSE mode uses 0.5 kbps Manchester encoded OOK detection which allows the chip to wake up upon detecting particular preamble and sync word pattern sent using OOK.

The transmitting node, which can be other than an EFR32xG22, must be configured to use the OOK based RFSENSE PHY for waking up the chip, via [RAIL_ConfigRfSenseSelectiveOokWakeupPhy](#), followed by setting the transmit FIFO to include the Preamble Byte and the sync word (1 byte - 4 bytes), using [RAIL_SetRfSenseSelectiveOokWakeupPayload](#). Note that the [RAIL_ConfigRfSenseSelectiveOokWakeupPhy\(\)](#) API is only supported on 2.4GHz chips and does not work on EFR32xG21 devices.

RAIL Multiprotocol

RAIL Multiprotocol

Overview

As of version 2.1, the RAIL library supports dynamic multiprotocol. In dynamic multiprotocol, an application can allocate multiple RAIL instances and configure them independently. Some radio operations are then arbitrated by a radio scheduler, which is internal to RAIL, to allow the different instances to coexist. The scheduler attempts to allow as many of these operations to run as possible by moving them around within the bounds allowed by the protocol.

Radio Scheduler

The radio scheduler is critical for multiprotocol operation. Its job is to arbitrate all radio operations, switch radio configurations, and ensure that radio operations run at the time requested. Not every RAIL function is a radio operation considered by the scheduler. Simple state update functions, such as [RAIL_SetTxPower\(\)](#), will directly change the hardware if active or update the state cache if inactive to have that change take affect the next time this protocol is loaded. The following RAIL functions are handled by the scheduler:

- [RAIL_StartRx\(\)](#)
- [RAIL_ScheduleRx\(\)](#)
- [RAIL_StartTx\(\)](#)
- [RAIL_StartScheduledTx\(\)](#)
- [RAIL_StartCcaCsmaTx\(\)](#)
- [RAIL_StartCcaLbtTx\(\)](#)
- [RAIL_StartAverageRssi\(\)](#)
- [RAIL_StartTxStream\(\)](#)

To arbitrate the radio operations, the scheduler uses `startTime`, `priority`, `slipTime`, and `transactionTime`. `startTime` comes implicitly from the RAIL API that started the transaction. For example, calling [RAIL_StartTx\(\)](#) will use a start time of right now while calling [RAIL_StartScheduledTx\(\)](#) takes an explicit start time as a parameter. The other three parameters are passed to all RAIL functions that require the scheduler in the [RAIL_SchedulerInfo_t](#) structure. The scheduler attempts to run each task at its desired `startTime`, but may also choose to run it up until `startTime + slipTime`. If it cannot run the task within that window, it will trigger a [RAIL_Config_t::eventsCallback](#) with the [RAIL_EVENT_SCHEDULER_STATUS](#) event set once that window has passed.

The scheduler is preemptive and will always ensure that a higher-priority task runs over a lower-priority task, but it does not guarantee ordering of these tasks. For example, it could choose to run a short lower-priority task with a small `slipTime` before a higher-priority task with a long `slipTime` to reduce dropped operations. Once an operation is started by the radio scheduler, it will either run to completion and trigger a normal RAIL event callback to indicate it is done or it will be aborted by a higher-priority task and ended with a [RAIL_EVENT_SCHEDULER_STATUS](#) event. Ultimately, the application is responsible for terminating most operations with a call to [RAIL_YieldRadio\(\)](#). This call cleans up internal state and allows the scheduler to switch to a lower priority operation.

Note

- For the [RAIL_SchedulerInfo_t::priority](#) parameter, 0 is the highest priority while 255 is the lowest.

Scheduler Operations

The APIs mentioned above can be classified as finite operations, infinite operations, and debug operations.

Finite Operations

Finite operations include the following APIs:

- [RAIL_ScheduleRx\(\)](#)
- [RAIL_StartTx\(\)](#)
- [RAIL_StartScheduledTx\(\)](#)
- [RAIL_StartCcaCsmaTx\(\)](#)
- [RAIL_StartCcaLbtTx\(\)](#)
- [RAIL_StartAverageRssi\(\)](#)

Finite operations make up the majority of the radio scheduler calls. A finite operation has a defined start time and either a defined or estimated end time. Each RAIL handle may only have one of these types of operations queued up at any time. The upper level application code is responsible for waiting until the previous operation has completed before starting a new one like in the single protocol RAIL library. If a new task is issued while a previous task is in progress, or if a higher-priority task interrupts an operation before the user has yielded, it will trigger the [RAIL_EVENT_SCHEDULER_STATUS](#) event with an appropriate [RAIL_SchedulerStatus_t](#) set and this operation will not be scheduled again.

Once a finite operation has begun, the scheduler will use the same priority until a new finite event is provided or a yield occurs. The scheduler will ensure that unnecessary switches do not occur if this next event is in the near future at the same or a higher priority. For more information see [Yielding the Radio](#).

Infinite Operations

Infinite operations include the following APIs:

- [RAIL_StartRx\(\)](#)

An infinite or background operation is treated by the scheduler as a radio state change. It still has a priority associated with it and will be started whenever it can be, but it will never fail. Another difference is that if this event is interrupted by a higher-priority task, it will be restarted whenever that task finishes where a finite task would be aborted.

Each RAIL handle may have only one infinite task but it may have both an infinite and finite or debug task at the same time. The one interesting aspect of this is that, if a finite task starts for this RAIL handle, the state will be updated to reflect what the infinite task has requested for that finite task's time period even if the infinite task is lower priority than another task in the system. For example, if you have one handle with a [RAIL_StartRx\(\)](#) of priority 5 and another handle with a [RAIL_StartRx\(\)](#) priority of 6 and a [RAIL_StartTx\(\)](#) priority of 3, the scheduler will choose the second handle's finite transmit task. While running this transmit task though, the receiver will be set to on so that, as soon as this transmit completes, the radio will transition from transmit into receive. It will remain in receive until the finite operation yields at which time the first handle's receive will take precedence.

This implementation is intended to get the radio to stay in receive for a very long time on one of the RAIL handles. It is generally only advisable to do this on one radio configuration at a time and for this to be the lowest-priority task. For well defined receive events, such as looking for an ACK scheduled receive, windows or state transitions will generally work better.

Note

- A known bug exists with turning on an infinite task during a finite task. For now, you must always configure the infinite task first to ensure it takes effect.
- If using state transitions that can lead from receive to idle, the user must tell the scheduler that this happened with a call to [RAIL_Idle\(\)](#). This restriction will hopefully be removed in future versions.

Debug Operations

Debug operations include the following APIs:

- [RAIL_StartTxStream\(\)](#)

Debug operations are somewhat special because they don't allow the user to pass a [RAIL_SchedulerInfo_t](#) structure to them. Instead, they implicitly provide a start time of now and use the highest possible priority. This means that they will have the highest priority and interrupt most other protocol operations to switch into the test mode. Nothing can preempt them

once started because the scheduler does not allow tasks of equal priority to preempt. These operations can be stooped by calling [RAIL_StopTxStream\(\)](#), [RAIL_YieldRadio\(\)](#), or [RAIL_Idle\(\)](#).

The modal nature of these debug operations is intentional to prevent multiple tone or stream operations from happening concurrently. It is expected that the user will manage these test modes from a higher level to prevent the different protocols from using tone or stream at the same time.

Yielding the Radio

Yielding is used by finite radio scheduler operations. Once an operation is started, that operation will run until explicitly yielded by the upper layer or interrupted by a higher-priority task. A yield is forced by a call to [RAIL_YieldRadio\(\)](#) or [RAIL_Idle\(\)](#). It is important that the application yield the radio as soon as it's done to allow lower-priority tasks to run.

Yielding is not handled by RAIL since terminating an operation can be application-specific. While RAIL may know that an individual transmit or receive operation is completed, there are sometimes other reasons to prevent a switch or to run a follow on operation. For example, if you are using ACKing you may want to wait for the [RAIL_EVENT_TXACK_PACKET_SENT](#) after a receive packet instead of yielding right after the receive. This also allows you to chain together many protocol-specific operations without the scheduler constantly context switching. If you do schedule a follow on operation that is far in the future, the scheduler will also be smart enough to treat this as a yield and slot in other lower-priority operations.

This does mean that you must carefully look through the [Events](#) in RAIL and subscribe to any that can impact your state machine. Specifically, things like transmit or receive success conditions, transmit and receive error conditions, and scheduler events ([RAIL_EVENT_SCHEDULER_STATUS](#), [RAIL_EVENT_CONFIG_UNSCHEDULED](#), and [RAIL_EVENT_CONFIG_SCHEDULED](#)) are important to keep track of. These would likely be needed in any upper layer to properly update the internal state machine as well so they're likely already in the implementation.

Building a Multiprotocol Application

The RAIL API is the same whether the targeted application is using the multiprotocol version or not. This was done intentionally to make switching between the two versions simple. From a build perspective, choose the desired RAIL library binary file to link into the application. For multiprotocol applications, the file name is `librail_multiprotocol_CHIP_COMPILER_release.a` while for other applications the file name is `librail_CHIP_COMPILER_release.a`.

Optionally, non multiprotocol applications can save memory by not allocating the [RAIL_Config_t::scheduler](#) state structure and passing NULL for all [RAIL_SchedulerInfo_t](#) pointers in APIs.

Starting in RAIL 2.12, the RAIL multiprotocol library internally provides two statically allocated RAM state buffers supporting two protocols, and both [RAILSched_Config_t](#) and [RAIL_StateBuffer_t](#) have been reduced to minimally-sized dummy structures solely for backwards compatibility. [RAIL_Config_t](#) structures no longer need to be in RAM or globals but can be allocated on the stack just for [RAIL_Init\(\)](#). In addition, [RAIL_Config_t::protocol](#) no longer needs to be provided for BLE (can always be NULL). If your multiprotocol application needs more than two protocols, additional state buffers for them must be provided by calling [RAIL_AddStateBuffer3\(\)](#) or [RAIL_AddStateBuffer4\(\)](#) prior to calling [RAIL_Init\(\)](#) for these protocols. If more than four protocols are needed, contact Silicon Labs for advice.

Example

Below is a simple example showing how to initialize two RAIL protocols and have one stay in infinite receive while the other one periodically transmits a packet at a higher priority.

```

#include <stdint.h>
#include <stdbool.h>
#include "rail.h"
#include "rail_config.h" // Generated by radio calculator

static RAIL_Handle_t gRailHandle1 = NULL;
static RAIL_Handle_t gRailHandle2 = NULL;

// Boolean to track when a packet has been sent.
static bool packetSendComplete = true;

// Creates transmit FIFOs for each PHY.
#define TX_FIFO_SIZE 128
static uint8_t txFifo1[TX_FIFO_SIZE];
static uint8_t txFifo2[TX_FIFO_SIZE];

static void radioConfigChangedHandler(RAIL_Handle_t railHandle,
                                     const RAIL_ChannelConfigEntry_t *entry)
{
    bool isSubgig = (entry->baseFrequency < 1000000000UL);

    // ... handle radio configuration change, e.g., select the desired PA possibly
    // using isSubgig to handle multiple configurations.
    RAIL_ConfigTxPower(railHandle, &railTxPowerConfig);

    // Reapply the Tx power after changing the PA above.
    RAIL_SetTxPowerDbm(railHandle, txPower);
}

static void radioEventHandler(RAIL_Handle_t railHandle,
                             RAIL_Events_t events)
{
    // Note that two different callbacks above could be used,
    // but this example only uses one which is split based on the handle.
    if (railHandle == gRailHandle1) {
        // Handle events for protocol 1.
        // NOTE: Since in infinite receive, the radio does not have to be yielded
        // here as below. If this were changed, the radio would have to be yielded here.
    } else if (railHandle == gRailHandle2) {
        // Handle any packet completion event (success or failure) and set us up
        // to send another packet.
        if (events & (RAIL_EVENT_TX_PACKET_SENT
                     | RAIL_EVENT_TX_ABORTED
                     | RAIL_EVENT_TX_UNDERFLOW
                     | RAIL_EVENT_SCHEDULER_STATUS)) {
            packetSendComplete = true;
            RAIL_YieldRadio(railHandle);
        }
    }
}

// Initializes the two PHY configurations for the radio.
void radioInitialize(void)
{
    // Creates each RAIL handle with their own configuration structures.
    RAIL_Config_t railCfg1 = {
        .eventsCallback = &radioEventHandler,
    };
    RAIL_Config_t railCfg2 = {
        .eventsCallback = &radioEventHandler,
    };
    gRailHandle1 = RAIL_Init(&railCfg1, NULL);
    gRailHandle2 = RAIL_Init(&railCfg2, NULL);

    // Sets up transmit FIFOs.
    RAIL_SetTxFifo(gRailHandle1, txFifo1, 0, TX_FIFO_SIZE);
    RAIL_SetTxFifo(gRailHandle2, txFifo2, 0, TX_FIFO_SIZE);
}

```

```
// Configures radio according to the generated radio settings.RAIL_ConfigChannels(gRailHandle1,
channelConfigs[0],&radioConfigChangedHandler);RAIL_ConfigChannels(gRailHandle2, channelConfigs[1],&radioConfigChangedHandler);//
Configures the most useful callbacks plus catch a few errors.RAIL_ConfigEvents(gRailHandle1,
    RAIL_EVENTS_ALL,
    RAIL_EVENT_TX_PACKET_SENT
    | RAIL_EVENT_TX_ABORTED
    | RAIL_EVENT_TX_UNDERFLOW
    | RAIL_EVENT_SCHEDULER_STATUS
    | RAIL_EVENT_RX_PACKET_RECEIVED
    | RAIL_EVENT_RX_FRAME_ERROR // invalid CRC | RAIL_EVENT_RX_ADDRESS_FILTERED);RAIL_ConfigEvents(gRailHandle2,
    RAIL_EVENTS_ALL,
    RAIL_EVENT_TX_PACKET_SENT
    | RAIL_EVENT_TX_ABORTED
    | RAIL_EVENT_TX_UNDERFLOW
    | RAIL_EVENT_SCHEDULER_STATUS
    | RAIL_EVENT_RX_PACKET_RECEIVED
    | RAIL_EVENT_RX_FRAME_ERROR // invalid CRC | RAIL_EVENT_RX_ADDRESS_FILTERED);}

int main(void){
    RAIL_SchedulerInfo_t schedulerInfo;
    uint32_t sendTime;// Initializes the radio and creates the two RAIL handles.radioInitialize();// Only set priority because transactionTime is
meaningless for infinite// operations and slipTime has a reasonable default for relative operations.
    schedulerInfo = (RAIL_SchedulerInfo_t){.priority = 200};RAIL_StartRx(gRailHandle1,0,&schedulerInfo);// Starts the first send 2 seconds after getting
set up.
    sendTime = RAIL_GetTime()+2000000;// Issues a transmit periodically on the second RAIL handle.while(true){if(packetSendComplete){
        RAIL_Status_t res;
        RAIL_ScheduleTxConfig_t scheduledTxConfig = {.when = sendTime,.mode = RAIL_TIME_ABSOLUTE };
        uint8_t packetData[]={0,1,2,3,4,5,6,7,7,8,10};// This assumes the Tx time is around 10 ms but should be tweaked based on// the specific PHY
configuration.
        schedulerInfo = (RAIL_SchedulerInfo_t){.priority = 100,.slipTime = 50000,.transactionTime = 10000};// Loads the transmit buffer with something to
send.RAIL_WriteTxFifo(gRailHandle2, packetData,sizeof(packetData),true);// Transmits this packet at the specified time or up to 50 ms late.
        res = RAIL_StartScheduledTx(gRailHandle2,0,
            RAIL_TX_OPTIONS_DEFAULT,&scheduledTxConfig,&schedulerInfo);if(res == RAIL_STATUS_NO_ERROR){
            packetSendComplete = false;
            sendTime += 2000000;// Add 2 seconds to the previous time.}else{// In the current configuration this should never
happen.assert(false);}}return 0;}
```

Understanding the Protocol Switch Time

Current EFR32 chips that run dynamic multiprotocol code only have a single radio in their hardware. As a result, when DMP switches protocols, it must fully reconfigure the radio accordingly. The time this reconfiguration takes, as well as the time it takes the scheduler to decide which radio task to run, and timings passed into RAIL via [RAIL_SetStateTiming](#) (idleToRx, idleToTx), make up the Protocol Switch Time. This is essentially the amount of time in advance the radio must be made aware of new tasks in order to execute them on time. If this time is not respected at higher layers, users may get a [RAIL_SchedulerStatus_t](#) that indicates a failed scheduled event. By default, RAIL uses [TRANSITION_TIME_US](#) as the time for all protocol switches. The following sections explain how to characterize this time for a specific app and reconfigure it.

Measuring the Protocol Switch Time

For users using only Silicon Labs provided standards-based stacks, the provided [TRANSITION_TIME_US](#) will be accurate. However, users implementing proprietary stacks or complex callbacks around scheduler events ([RAIL_EVENT_CONFIG_UNSCHEДУLED](#), [RAIL_EVENT_CONFIG_SCHEDULED](#), and [RAIL_EVENT_SCHEDULER_STATUS](#)) may need to recharacterize this time if they find that the current switch time is causing scheduling failures in their application. Users can empirically determine this time for their system with the following algorithm: On the EFR, set a small (e.g. 0) transition time using [RAIL_SetTransitionTime](#) (this API must be called before [RAIL_Init](#) is called on any protocols). On protocol 1, enter infinite RX with [RAIL_StartRx](#). On protocol 2, request an absolutely scheduled TX for some time in future (e.g. `RAIL_GetTime() + 1000000`). For small transition time values, the application should get [RAIL_EVENT_SCHEDULER_STATUS](#) on protocol 2, with a scheduler status of [RAIL_SCHEDULER_STATUS_SCHEDULED_TX_FAIL](#). Repeat this test (resetting the device each time between runs to ensure [RAIL_SetTransitionTime](#) is called before [RAIL_Init](#)) with increasing transition times. Eventually, for some large enough value, the application should begin getting [RAIL_EVENT_TX_PACKET_SENT](#) events, indicating that the supplied transition

time was sufficient, and thus the radio was able to transition in time to complete the scheduled transmit at the correct time. Ideally this test should be run in both directions (i.e. repeat the test, swapping protocols 1 and 2).

Sample code for the above algorithm is provided below:

```

#include <stdint.h>
#include <stdbool.h>
#include "rail.h"
#include "rail_config.h" // Generated by radio calculator

static RAIL_Handle_t gRailHandle1 = NULL;
static RAIL_Handle_t gRailHandle2 = NULL;

// Boolean to track when a packet has been sent.
static bool packetSendComplete = true;

// Creates transmit FIFOs for each PHY.
#define TX_FIFO_SIZE 128
static uint8_t txFifo1[TX_FIFO_SIZE];
static uint8_t txFifo2[TX_FIFO_SIZE];

uint32_t transitionTime = 0;

void success(void) {
    // Indicate test success to user
}

void fail(void) {
    // Indicate test failure to user
}

static void radioConfigChangedHandler(RAIL_Handle_t railHandle,
    const RAIL_ChannelConfigEntry_t *entry)
{
    bool isSubgig = (entry->baseFrequency < 1000000000UL);

    // ... handle radio configuration change, e.g., select the desired PA possibly
    // using isSubgig to handle multiple configurations.
    RAIL_ConfigTxPower(railHandle, &railTxPowerConfig);

    // Reapply the Tx power after changing the PA above.
    RAIL_SetTxPowerDbm(railHandle, txPower);
}

static void radioEventHandler(RAIL_Handle_t railHandle,
    RAIL_Events_t events)
{
    // Note that two different callbacks above could be used,
    // but this example only uses one which is split based on the handle.
    if (railHandle == gRailHandle1) {
        // We don't care about events on protocol 1 for this test
        return;
    } else if (railHandle == gRailHandle2) {
        // Handle any packet completion event (success or failure) and set us up
        // to send another packet.
        if (events & RAIL_EVENT_TX_PACKET_SENT) {
            success();
        }
        if ((events & RAIL_EVENT_SCHEDULER_STATUS)
            && (RAIL_GetSchedulerStatus(railHandle) & RAIL_SCHEDULER_STATUS_SCHEDULED_TX_FAIL)) {
            fail();
        }
    }
}

// Initializes the two PHY configurations for the radio.
void radioInitialize(void)
{
    // transitionTime should be swept via testing instrumentation between
    // runs
    RAIL_SetTransitionTime(transitionTime)

```

```
// Creates each RAIL handle with their own configuration structures.
RAIL_Config_t railCfg1 = {eventsCallback = &radioEventHandler,};
RAIL_Config_t railCfg2 = {eventsCallback = &radioEventHandler,};
gRailHandle1 = RAIL_Init(&railCfg1, NULL);
gRailHandle2 = RAIL_Init(&railCfg2, NULL); // Sets up transmit FIFOs. RAIL_SetTxFifo(gRailHandle1, txFifo1, 0,
TX_FIFO_SIZE); RAIL_SetTxFifo(gRailHandle2, txFifo2, 0, TX_FIFO_SIZE); // Configures radio according to the generated radio
settings. RAIL_ConfigChannels(gRailHandle1, channelConfigs[0], &radioConfigChangedHandler); RAIL_ConfigChannels(gRailHandle2,
channelConfigs[1], &radioConfigChangedHandler); // Only need a few events on the second handle for this test RAIL_ConfigEvents(gRailHandle2,
    RAIL_EVENTS_ALL,
    RAIL_EVENT_TX_PACKET_SENT
    | RAIL_EVENT_SCHEDULER_STATUS);}

int main(void){
    RAIL_SchedulerInfo_t schedulerInfo;
    uint32_t sendTime; // Initializes the radio and creates the two RAIL handles. radioInitialize(); // Get into receive on handle 1
    schedulerInfo = (RAIL_SchedulerInfo_t){.priority = 200, .transactionTime = 1000, .slipTime = 0}; RAIL_StartRx(gRailHandle1, 0, &schedulerInfo); // Starts
the first send 1 second after getting set up.
    sendTime = RAIL_GetTime() + 1000000; // Request a TX for some time in the future
    RAIL_ScheduleTxConfig_t scheduledTxConfig = {.when = sendTime, .mode = RAIL_TIME_ABSOLUTE };

    schedulerInfo.priority = 100;
    res = RAIL_StartScheduledTx(gRailHandle2, 0,
        RAIL_TX_OPTIONS_DEFAULT, &scheduledTxConfig, &schedulerInfo); return 0;}
```

Minimizing the Protocol Switch Time

Currently, the best way to minimize the switch time is by proper design of [RAIL_Config_t::eventsCallback](#). Specifically, the three scheduler events [RAIL_EVENT_CONFIG_UNSCHEDULED](#), [RAIL_EVENT_CONFIG_SCHEDULED](#), and [RAIL_EVENT_SCHEDULER_STATUS](#), will be passed to the callback independently of all other events. Additionally, they are only raised during protocol switch process, and so their handling is part of the critical path for the protocol switch. Ideally, if they are not needed, [RAIL_ConfigEvents](#) should disable these events. Otherwise they should be handled first, after which the event handler should return immediately.

Additionally, short transition times should be specified in [RAIL_SetStateTiming](#). Note that 0 for these values can be unreliable in a DMP context, as that tells the radio to go "as fast as possible" as opposed to a reliable, known value.

Device Information Utility

Device Information Utility

RAIL Utility for Reading Device Information

This software component allows reading and storing device specific information from the device to be used later for optimal radio performance.

This component must be included only for Silicon Labs Series 3 devices and must be initialized before calling [RAIL_Init\(\)](#) outside a critical section.

PA Power Configs in persistent memory

PA Power Configs in persistent memory

RAIL PA NVM Configs Utility

This optional software component can be enabled to allow overriding an application's compile-time default configurations for PA mode and PA power curve/table settings for the supported modes. This might be used to calibrate a particular device or set of devices. These overriding settings must still conform to the platform's possible PA modes and curve/table sizes. This component currently supports only EFR32xG24 and xG26.

By enabling this component, at application startup time the [Power Amplifier \(PA\) Utility](#) component will check for the existence of these NVM3 tokens:

- [SL_RAIL_UTIL_PA_NVM_MODE_TAG](#) containing a [RAIL_TxPowerMode_t](#).
If this token exists and is a valid PA mode supported by the platform, it will override the compile-time [RAIL_TxPowerConfig_t::mode](#) returned by [sl_rail_util_pa_get_tx_power_config_2p4ghz\(\)](#) which is used by stacks and applications to get the [Power Amplifier \(PA\) Utility](#) component's power configuration.
- [SL_RAIL_UTIL_PA_NVM_CONFIG_TAG](#) containing a [sl_rail_nvm_pa_config_t](#).
If this token exists and is valid, it will override the compile-time [RAIL_TxPowerCurvesConfigAlt_t](#) configuration which is applied in [Power Amplifier \(PA\) Utility](#) component's initialization call to [RAIL_InitTxPowerCurvesAlt\(\)](#).

The component provides APIs for writing and reading these NVM3 tokens, please refer to the [sl_rail_util_pa_nvm_configs.h](#) file for details.

See these documents for in-depth explanations of how to configure and use the available PA features:

- AN1127: Power Amplifier Power Conversion Functions in RAIL 2.x

Configuration Options

This component has no configuration options.

EFR32xG24 Example

The default PA curves/tables configuration that RAIL provides for the VBAT powered 20 dBm EFR32xG24 (in `plugin/pa-conversions/efr32xg24/sl_rail_util_pa_curves_20dbm.h`) would have the following [sl_rail_nvm_pa_config_t](#) layout when stored in the NVM3 [SL_RAIL_UTIL_PA_NVM_CONFIG_TAG](#). Multi-byte fields must be stored in little-endian format. The byte offset of each field/value is shown in comments below:

```

static sl_rail_nvm_pa_config_t serialized_pa_config = {
/*byte*/
/* 0*/ .version      = 1,
/* 1*/ .num_descriptors = 2,
/* 2*/ .pa_voltage    = 0,
/* 4*/ .signature     = 0,
/* 8*/ .pa_descriptors = {
/* 8*/ { .algorithm      = 0,
/* 9*/ .num_segments_or_entries = 8,
/* 10*/ .min             = 0,
/* 11*/ .max             = 180,
/* 12*/ },
/* 12*/ { .algorithm      = 1,
/* 13*/ .num_segments_or_entries = 16,
/* 14*/ .min             = 0,
/* 15*/ .max             = 15,
/* 16*/ },
/* 16*/ },
/* 16*/ .pa_curve_or_table = {
/* 16*/ { .curve = {
/* 16*/ .curve_min_ddbm = -338,
/* 18*/ .curve_max_ddbm = 200,
/* 20*/ .curve_segments = {
/* 20*/ { .maxPowerLevel = 180,
/* 22*/ .slope          = 2280,
/* 24*/ .intercept      = -291457,
/* 28*/ },
/* 28*/ { .maxPowerLevel = 78,
/* 30*/ .slope          = 770,
/* 32*/ .intercept      = -46749,
/* 36*/ },
/* 36*/ { .maxPowerLevel = 44,
/* 38*/ .slope          = 431,
/* 40*/ .intercept      = -6673,
/* 44*/ },
/* 44*/ { .maxPowerLevel = 27,
/* 46*/ .slope          = 255,
/* 48*/ .intercept      = 6886,
/* 52*/ },
/* 52*/ { .maxPowerLevel = 17,
/* 54*/ .slope          = 167,
/* 56*/ .intercept      = 10458,
/* 60*/ },
/* 60*/ { .maxPowerLevel = 10,
/* 62*/ .slope          = 98,
/* 64*/ .intercept      = 10261,
/* 68*/ },
/* 68*/ { .maxPowerLevel = 6,
/* 70*/ .slope          = 59,
/* 72*/ .intercept      = 8616,
/* 76*/ },
/* 76*/ { .maxPowerLevel = 3,
/* 78*/ .slope          = 11,
/* 80*/ .intercept      = 3745,
/* 84*/ },
// : a 9th curve segment is allowed but not used so this is pad
/* 92*/ },
/* 92*/ },
/* 92*/ },
/* 92*/ { .table = {
/* 92*/ { -250,
/* 94*/ -148,
/* 96*/ -95,
/* 98*/ -68,
/* 100*/ -51,
/* 102*/ -40,
/* 104*/ -32,

```

can be substituted for a table so this is $\text{pad}^* \{168^*\}_{i,j} / \{168^*\}_{i,j}$

Release Notes

Release Notes

RAIL Library 2.19.0 GA

June 16, 2025

Release Highlights

- Added support for Concurrent Listening in combination with IEEE 802.15.4 High Data Rate (1 Mbps and 2 Mbps custom 802.15.4 SUN PHYs) on the EFR32xG26 part.
- Added support for EFR32xG29 part.
- Added support for EFR32xG22L part.
- Added support for storing Antenna calibrations like RSSI offset, PA mode, PA curves/tables in NVM3 on EFR32xG24 and EFR32xG26 parts.
- Added support for the EFR32xG24 SiP Module.
- Bugfixes and minor improvements.

Deprecation Notes

- The RAIL 2.x API is planned for deprecation in the 25Q4-GA release (December 2025). At that time, the new RAIL 3.0 API will be released for all supported chips along with a RAIL 2.x compatibility layer and migration guide.
 - The goal of this new API is to get rid of some unused features, add better support for concurrent listening use cases, and to simplify channel and PA configurations.
 - The migration is intended to be straight forward and simple for the majority of customers. However, in some cases manual help might be required to ease this transition.
- Note: RAIL 3.0 API is in evaluation in sl_rail_*.h files, though subject to change, and the RAIL library has RAIL 3.0 symbols alongside RAIL 2.x, with some stacks and RAIL apps having started to use it in 25Q2-GA.

Release Details

- [API Changes](#)
- [RAIL Changelog](#)
- [RAILtest Changelog](#)
- [Deprecated List](#)

Modules

[RAIL Changelog](#)

[RAILtest Changelog](#)

[API Changes](#)

RAIL Changelist

RAIL Changelist

2.19.0

- **1274214:** Fixed an issue on EFR32xG23, EFR32xG25, EFR32xG26 and EFR32xG28 where radio was blocked in [RAIL_RF_STATE_RX_ACTIVE](#).
- **1355305:** Fixed an issue where the events [RAIL_EVENT_RX_PREAMBLE_DETECT](#), [RAIL_EVENT_RX_PREAMBLE_LOST](#) and [RAIL_EVENT_RX_SYNC1_DETECT](#) were not generated for WiSUN OQPSK PHYs on the EFR32xG25 part.
- **1356085:** The rail_util_rssi component can now store and retrieve the chip RSSI offset from NVM3 persistent storage.
- **1357024:** Fixed a false CCA busy issue on the EFR32xG28 part.
- **1357942:** Added new component [PA Power Configs in persistent memory](#) providing ability to store PA mode and/or PA curves/tables configs in persistent NVM3 on EFR32xG24 and EFR32xG26, overriding their compile-time configurations.
- **1414408:** Added RAIL support for the MGM270SC32SNA2 module.
- **1425355:** Updated documentation to clarify the usage of [RAIL_TX_OPTION_CCA_PEAK_RSSI](#) which can result in false channel clear determinations.
- **1426472:** Fixed an issue where PTI was only partially disabled when configured with [RAIL_PTI_MODE_DISABLED](#) (which is RAIL's default), and accordingly clarified [RAIL_ConfigPti\(\)](#) and [RAIL_EnablePti\(\)](#) documentation.
- **1435514:** Fixed an issue on the EFR32xG24, where enabling channel sounding under certain conditions can potentially result in hanging the system in the default handler.
- **1436580:** Clarified [RAIL_SetRssiOffset\(\)](#) documentation, describing the range of valid values for radio and protocol offsets.
- **1437972:** Fixed an issue on the EFR32xG24, where the RSSI reported on certain channel sounding SYNC packets could be lower than expected.
- **1439738:** Updated the documentation to clarify that [RAIL_GetChannelHoppingRssi\(\)](#) returns the RSSI in units of quarter-dBm and not deci-dBm.
- **1439816:** Fixed an issue with Wi-SUN mode switch where idling the radio after [RAIL_EVENT_IEEE802154_MODESWITCH_START](#) can prevent reception of future mode switch headers.
- **1445964:** Fixed an issue in [RAIL_GetRxPacketDetails\(\)](#) where the duration of the packet would be incorrect when packet size is larger than 534 bytes on the EFR32xG25 part.
- **1450906:** Fixed an issue on the EFR32xG24, where CS antenna switching would not work when configured for GPIOs on PortC or PortD.

2.18.2

- **1375717:** Updated component documentation for [RF Path Switch Utility](#) and [RF Path Utility](#) to clarify appropriate usage.
- **1389592:** Fixed an issue in [Coexistence Utility](#) where enabling Grant, Priority and Request signals would cause hard fault in Matter + BLE multiprotocol application.
- **1412900:** Fixed an issue in the [BLE Channel Sounding \(CS\) Antenna Offset Utility](#) where array lengths of wired and wireless antenna offset values were fixed to a default size regardless of the antenna count of a selected board.
- **1419786:** Updated Wi-SUN mode switch channel mask for BZ band in RAILTEST on the EFR32xG25 part.
- **1426191:** Fixed an issue in dynamic multiprotocol applications where a radio does not enter into EM2 sleep mode when no active tasks are being performed.

2.18.1

- **1172597:** Fixed an issue with [RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL](#) where [RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE](#) wouldn't be signaled. Added new [RAIL_TriggerRxChannelHop\(\)](#) API for manually triggering an RX channel hop, a refinement of [RAIL_CalibrateTemp\(\)](#). Also noted [RAIL_RX_CHANNEL_HOPPING_OPTION_STOP](#) was only intended for testing purposes and is not supported on the EFR32xG21 part.
- **1349369:** Fixed an issue where power restrictions were incorrectly applied through channel configs on the platforms that support [RAIL_SUPPORTS_DBM_POWERSETTING_MAPPING_TABLE](#).

- **1365389:** Added RAIL support for the MGM260PD32VNA2, MGM260PD32VNN2, MGM260PD22VNA2, MGM260PB32VNA5, MGM260PB32VNN5, MGM260PB22VNA5, BGM260PB22VNA2 and BGM260PB32VNA2 modules.
- **1380153:** Fixed an issue where calling [RAIL_Idle\(\)](#) can prevent any future reception on the EFR32xG25 for OFDM-FSK concurrent PHYs.
- **1389189:** Fixed a false channel busy issue when automatic LNA bypass is enabled on the EFR32xG25 part.

2.18.0

- **748807:** Added [RAIL_GetRxIncomingPacketRssi](#) to get the RSSI associated with the incoming packet.
- **1120643:** Fixed an issue where the phy was not re-configured correctly when using the [RAIL_ConfigSyncWords](#) API with certain custom modem configurations.
- **1269262:** Added support for [BLE Channel Sounding \(CS\) Antenna Offset Utility](#) for BLE Channel Sounding feature on the EFR32xG24 that allows customer to configure offsets for a wired or wireless antenna configuration.
- **1273595:** To better reflect Z-Wave spec naming conventions, renamed [RAIL_ZWAVE_REGION_US_LR_END_DEVICE](#) to [RAIL_ZWAVE_REGION_US_LR3](#), [RAIL_ZWAVE_REGION_EU_LR_END_DEVICE](#) to [RAIL_ZWAVE_REGION_EU_LR3](#), [RAIL_ZWAVE_REGIONID_US_LR_END_DEVICE](#) to [RAIL_ZWAVE_REGIONID_US_LR3](#), and [RAIL_ZWAVE_REGIONID_EU_LR_END_DEVICE](#) to [RAIL_ZWAVE_REGIONID_EU_LR3](#).
- **1290406:** To prepare for RAIL 3.0 specifications, [RAIL_Status_t](#) is now [sl_status_t](#) and [RAIL_STATUS_](#) values are mapped to similar [SL_STATUS_](#) values.
- **1290731:** Fixed a multiprotocol issue where [RAIL_BLE_SetNextTxRepeat\(\)](#) or [RAIL_SetNextTxRepeat\(\)](#) information mistakenly survived across [RAIL_Idle\(\)](#) to the surprise of a subsequent non-repeated transmit.
- **1300509:** Fixed an issue with IEEE 802.15.4 High Data Rate (HDR) packets such that PTI now properly indicates they have a 2-byte 802.15.4 PHY header on the EFR32xG26 part.
- **1306804:** Fixed an issue where glitches could occur when using [RAIL_RxDataSource_t](#) values other than [RX_PACKET_DATA](#) on the EFR32xG23 and newer chips.
- **1309139:** Fixed an issue on the EFR32xG22 and later where [RAIL_StopTx\(\)](#) being called before the transmit has started could cause the radio to return to receive with the radio partially misconfigured for transmit.
- **1309208:** Fixed a minor issue on the EFR32xG21 where the TX-to-RX turnaround timing could be a bit longer than specified.
- **1324392:** Added support for TrustZone APIs (see [api_changes_2_18_0](#) for more details) on the EFR32xG23 and EFR32xG28 parts to allow RAIL to access the CMU, EMU, GPIO, LDMA, HFXO, PRS and SYSRTC peripherals when they are running in the secure state despite the RAIL library being non-secure.
- **1338958:** Fixed documentation of [RAIL_PacketTimeStamp_t::packetDurationUs](#) which is supported on all platforms for normal transmits but no platforms for Auto-Ack transmits.
- **1352393:** Fixed an issue on ZGM230S modules where PHY signatures were computed incorrectly and the device would assert on startup.
- **1356059:** Updated the EFR32xx documentation to better clarify the RAIL dependencies on HFXO and TrustZone peripheral configuration.
- **1357948:** Fixed a DMP issue where the device would incorrectly switch to a background receive task with insufficient time to switch back to an upcoming high priority scheduled task.
- **1369901:** Fixed an issue where the device would become stuck when running image rejection calibration on an OFDM PHY from the event callback on the EFR32xG25 part.

2.17.2

- **1276280:** Fixed a DMP scheduling issue where tasks were scheduled to run with inadequate time to complete before another pending task.
- **1306597:** Fixed a CCA issue on EFR32xG28 where, in some cases, the channel could be reported as busy even though it was free.
- **1329239:** Reduced the delay between TX completion and the start of PA ramp down on the EFR32xG24.

2.17.1

- **1271542:** Fixed an issue on EFR32xG21 where an inopportune-timed [RAIL_StopTx\(\)](#) using [RAIL_STOP_MODE_ACTIVE](#) can hang a subsequent CSMA/LBT transmit during its clear-channel assessment (CCA).
- **1276280:** Fixed a scheduling issue where tasks were scheduled to run with inadequate time to complete before another pending task.
- **1325463:** Added support of automatic LNA bypass on the EFR32xG25 parts.

2.17.0

- **758067:** Added support of collision detection for concurrent PHYs on the EFR32xG25 parts.
- **1227935:** Added support for additional Coex TX and RX metrics events to the [Coexistence Utility](#) component.
- **1242662:** Fixed incorrect default values for SL_RAIL_UTIL_PA_CALIBRATION_ENABLE on the EFR32xG24 and EFR32xG25 parts, along with an issue where Power Amplifier calibrations were not properly disabled when SL_RAIL_UTIL_PA_CALIBRATION_ENABLE was disabled.
- **1247525:** Updated few APIs to return [RAIL_Status_t](#) which were returning void previously.
- **1247525:** Updated [RAIL_IEEE802154_WriteEnhAck](#) and [RAIL_WriteAutoAckFifo\(\)](#) which formerly took uint8_t ackDataLen parameter now take uint16_t ackDataLen parameter.
- **1247525:** Added missing runtime equivalent API [RAIL_SupportsPathDiversity\(\)](#) of [RAIL_SUPPORTS_PATH_DIVERSITY](#).
- **1251783:** Removed documentation pertaining to EFR32 Series-1 platforms and updated chip-specific documentation for Series-2 platforms.
- **1255524:** Fixed a dynamic multiprotocol issue where [RAIL_BLE_Init\(\)](#) was called (which implicitly establishes the BLE 1Mbps PHY), but after a protocol switch returns to BLE, its [RAIL_BLE_ConfigChannelRadioParams\(\)](#) settings were not being properly reapplied.
- **1256045:** Fixed an issue where Signal Identifier options were being errantly displayed in Simplicity Studio for the [Coexistence Utility](#) component on platforms that don't support the Signal Identifier feature.
- **1265376:** Fixed an issue in the UC components where the RAIL power manager support could be initialized after the radio was started in certain example applications. This would lead to radio state and power mode requirements being improperly tracked as you must initialize power manager support before any radio operations.
- **1266003:** Fixed an issue where [RAIL_BLE_ConfigChannelRadioParams\(\)](#) was improperly overriding the BLE Quuppa PHY whitening configuration.

2.16.2

- **1248013:** Fixed an issue with IEEE 802.15.4 High Data Rate (HDR) packets such that PTI now properly indicates they have a 2-byte 802.15.4 PHY header.
- **1255347:** Fixed an issue with [RAIL_SetTxFifo\(\)](#) setup that could trip [RAIL_ASSERT_FAILED_UNEXPECTED_STATE_TX_FIFO](#) when the prior TX FIFO was larger and contained more data than the new smaller TX FIFO can hold. Any data in the old TX FIFO is now properly ignored.
- **1271435:** Fixed an issue where [RAIL_WriteTxFifo\(\)](#) could write outside the TX FIFO when called without resetting the FIFO in dynamic multiprotocol applications.

2.16.1

- **1161746:** Updated [Wireless M-Bus](#) protocol documentation and grouped it under [Protocol-specific](#) hardware acceleration section of RAIL API document.
- **1168174:** Added support for setting the Whitening and CRC initial values at runtime that are defined by the existing PHY's radio configuration.
- **1241800:** AFixed an issue where we were missing the pa_dbm_mapping_table.py script in the GSDK. This script is used to help create power tables on the EFR32xG25 part.
- **1242723:** Fixed a situation where interrupts could be left off and the system would hang when using dynamic multiprotocol on the EFR32xG25.
- **1243727:** Improved the CCA reliability on the EFR32xG23, EFR32xG25 and EFR32xG28 chips.

2.16.0

- **663884:** Added better information for [Front End Module \(FEM\) Utility](#) component and its usage.
- **689770:** Added support for a new assert, which will be thrown if the loaded PHY is not supported by the software defined modem on the EFR32xG25.
- **748817:** Added new [RAIL_GetAutoAckFifo\(\)](#) API and allow NULL for [RAIL_WriteAutoAckFifo\(\)](#) or [RAIL_IEEE802154_WriteEnhAck\(\)](#) ackData parameter, which gives applications direct access to the AutoAck FIFO to construct Ack packets in pieces.
- **751902:** Updated the default PTI rate to 3200000 bps on the EFR32xG25.

- **858169:** Updated documentation on the usage of [RAIL_EVENT_RX_TIMING_DETECT](#), [RAIL_EVENT_RX_TIMING_LOST](#), [RAIL_EVENT_RX_PREAMBLE_DETECT](#) and [RAIL_EVENT_RX_PREAMBLE_LOST](#) events and the possible issues that may be encountered when using these with certain demodulators on EFR32xG2x chips.
- **1079816:** Fixed a race condition on the EFR32xG22 and later during RX channel hopping or duty-cycling where frame detection occurring close to when a hop should happen could leave the radio stuck in reception but not receiving anything, with the only remedy being to idle the radio.
- **1088439:** Fixed an issue which would cause the incorrect antenna to be reported for a received packet on the EFR32xG25 when using OFDM and antenna diversity.
- **1104367:** Added support for antenna selection through the applicable [RAIL_RxOptions_t](#) and [RAIL_TxOptions_t](#) values when using OFDM on the EFR32xG25.
- **1112324:** Added a new [Software Modem \(SFM\) sequencer image selection Utility](#) component to allow selection of modulations supported by EFR32xG25 software modem (SFM). These changes can save considerable flash space by reducing the set of modulations to just those that are needed.
- **1117349:** No longer include `rail_chip_specific.h` and `rail_features.h` (anything dependent on `em_device.h`) when `SLI_LIBRARY_BUILD` is defined. This will allow the user to build their radio code in a way that depends on RAIL generically but is not chip-specific. When doing this the code cannot depend on things that are inherently chip-specific and are still in those files like `RAIL_RF_PATHS`, `RAIL_NUM_PA`, or any of the compile-time `RAIL_SUPPORTS_xxx` defines. Code will need to call appropriate runtime APIs or split itself between the generic and chip specific portions and build them separately.
- **1117349:** To better support chip-agnostic builds [RAIL_TxPowerMode_t](#) is now a superset representing all possible PAs across all chips. Any code depending on the number of or consecutive ordering of chip-supported PAs will likely need to be updated.
- **1117349:** To better support chip-agnostic builds [RAIL_CalValues_t](#) and subordinate [RAIL_IrCalValues_t](#) have grown to encompass the superset of fields needed on all chips, affecting all chips except EFR32xG25.
- **1117349:** To better support chip-agnostic builds [RAIL_TransitionTime_t](#) and therefore [RAIL_StateTiming_t](#) have grown to a superset type, affecting EFR32xG1.
- **1117349:** To better support chip-agnostic builds [RAIL_FIFO_ALIGNMENT](#) is now universally 32-bit, affecting EFR32xG1x and EFR32xG21, but is still only enforced on chips actually requiring that alignment.
- **1129262:** Added support for Sidewalk PHYs on the EFR32xG23 and EFR32xG28 chips.
- **1129280:** Added an assert [RAIL_ASSERT_INVALID_XTAL_FREQUENCY](#) on the EFR32xG1x and EFR32xG2x chips, to fire when a radio config that is loaded is incompatible with a device due to the defined crystal frequency of the config not matching with the device's crystal frequency.
- **1131398:** Added the [RAIL_TX_REPEAT_OPTION_START_TO_START](#) option to measure the delay between repeated transmits from the start of TX to start of TX instead of the default from end of TX to start of TX.
- **1153679:** Fixed an issue in [Coexistence Utility](#) on the EFR32xG24 where a GRANT signal pulse less than 100us might result in the radio not being properly placed in hold off after GRANT is deasserted.
- **1156980:** Fixed an issue with channel hopping on the EFR32xG22 and later where use of [RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD](#) can prevent the timed RX channel hopping modes including RX duty-cycling from timing out properly.
- **1157301:** Added support for GCC 12.2.1 and IAR 9.40.1 compilers.
- **1159773:** Added support for Fast Channel Switching PHYs on the EFR32xG24.
- **1161043:** Added support for [RAIL_IEEE802154_SUPPORTS_G_MODESWITCH](#) on the EFR32xG28.
- **1163285:** Added support for the IEEE802154 2.4 GHz coherent PHYs via [RAIL_SUPPORTS_IEEE802154_BAND_2P4](#) on the EFR32xG28.
- **1171683:** Added a new [RAIL_RxOptions_t](#) option to enable collision detection on the EFR32xG25. Once enabled, when a collision with a strong enough packet is detected, the demod will stop the current packet decoding and try to detect the preamble of the incoming packet.
- **1174185:** Updated the documentation of [RAIL_RxChannelHoppingMode_t::RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL](#) and its current limitation of not issuing [RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE](#).
- **1175032:** Updated the documentation mentioning that while configuring DPLL on all Series 2 chips, Phase Lock mode should typically be used instead of Frequency Lock mode.
- **1175684:** Fixed an issue with the [RAIL_IDLE](#) form of [RAIL_Idle\(\)](#) and [RAIL_STOP_MODE_PENDING](#) form [RAIL_StopTx\(\)](#) during the LBT/CSMA phase of a transmit which could previously hang. Pending LBT/CSMA and scheduled transmits are now stopped or idled with [RAIL_EVENT_TX_BLOCKED](#) triggered, except for idle mode [RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS](#).
- **1183040:** Updated all PHYs built into RAIL on Series 2 platforms using the latest radio calculator to reduce observed high reference spurs.

- **1184982:** Fixed an issue with [RAIL_StartAverageRssi\(\)](#) that could cause execution to hang in interrupt context just before raising [RAIL_EVENT_RSSI_AVERAGE_DONE](#). This was primarily an issue on the EFR32xG21 platform. Note that it is possible, though unlikely, that [RAIL_GetAverageRssi\(\)](#) might still return [RAIL_RSSI_INVALID](#) after the average RSSI period has finished.
- **1184982:** Fixed an issue with [RAIL_StartAverageRssi\(\)](#) that caused it to mistakenly idle the radio in the newly-activated protocol if a dynamic protocol switch occurred during the averaging operation of the suspended protocol.
- **1187168:** Updated the default power curves for the 10dBm High Power PA on EFR32xG24.
- **1188083:** Fixed an issue where [RAIL_Idle\(\)](#) could hang waiting for the radio to idle when using IEEE 802.15.4 fast RX channel switching.
- **1190187:** Fixed an issue where idling the radio before a Scheduled Receive window ends could cause a subsequent packet that should be silently filtered and rolled back to instead be received with [RAIL_RX_PACKET_READY_CRC_ERROR](#).
- **1196683:** Added support of 802.15.4 IMM-ACK with OFDM and OQPSK modulations (FCS 4 bytes only) on the EFR32xG25.
- **1201506:** Fixed an issue in multiprotocol applications where the wrong sync word would be used if two protocols used the same radio configuration and only one of those protocols set a custom sync word with [RAIL_ConfigSyncWords\(\)](#) API.
- **1211224:** Updated the RAIL Library component to include the [Built-in PHYs for alternate HFXO frequencies](#) component automatically to better support different HFXO frequencies on supported parts.

2.15.2

- **634068:** Reduced the delay between TX completion and the start of PA ramp down on the EFR32xG24.
- **663884:** Added better information for [Front End Module \(FEM\) Utility](#) component and its usage.
- **689770:** Added support for a new assert, which will be thrown if the loaded PHY is not supported by the software defined modem on EFR32xG25.
- **1131398:** Added the [RAIL_TX_REPEAT_OPTION_START_TO_START](#) option to measure the delay between repeated transmits from the start of TX to start of TX instead of the default from end of TX to start of TX.
- **1191666:** Fixed an issue where high reference spurs may appear during radio communication on EFR32xG22 and newer platforms.

2.15.1

- **707731:** Fixed an issue when using [RAIL_BLE_SetNextTxRepeat\(\)](#) that would cause an incorrect Protocol Config ID to appear in the packet trace for repeated transmits.
- **748817:** Added new [RAIL_GetAutoAckFifo\(\)](#) API and allow NULL for [RAIL_WriteAutoAckFifo\(\)](#) or [RAIL_IEEE802154_WriteEnhAck\(\)](#) ackData parameter, which gives applications direct access to the AutoAck FIFO to construct Ack packets in pieces.
- **1079816:** Fixed a race condition on EFR32xG22 and later during RX channel hopping or duty-cycling where frame detection occurring close to when a hop should happen could leave the radio stuck in reception but not receiving anything, with the only remedy being to idle the radio.
- **1088439:** Fixed an issue which would cause the incorrect antenna to be reported for a received packet on the EFR32xG25 when using OFDM and antenna diversity.
- **1097499:** Updated the [RAIL_PacketTimeStamp_t::packetDurationUs](#) field on the EFR32xG25 to be populated for non-OFDM packets.
- **1104367:** Added support for antenna selection through the applicable [RAIL_RxOptions_t](#) and [RAIL_TxOptions_t](#) values when using OFDM on the EFR32xG25.
- **1112324:** Added a new [Software Modem \(SFM\) sequencer image selection Utility](#) component to allow selection of modulations supported by EFR32xG25 software modem (SFM). These changes can save considerable flash space by reducing the set of modulations to just those that are needed.
- **1129262:** Added support for Sidewalk PHYs for EFR32xG23 and EFR32xG28 chips.
- **1141539:** Fixed an issue to improve CCA behavior for regional certification test suites for EFR32xG25 chips.
- **1153679:** Fixed an issue in [Coexistence Utility](#) where a GRANT signal pulse less than 100us might result in the radio not being properly placed in hold off after GRANT is deasserted.
- **1153921:** Added a new component to switch between Coexistence, Antenna Diversity and FEM Utilities for EFR32xG21 and EFR32xG24 chips supporting 15.4 Fast Channel Switching feature.
- **1156980:** Fixed an issue with channel hopping on EFR32xG22 and later where use of [RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD](#) can prevent the timed RX channel hopping modes including RX duty-cycling from timing out properly.
- **1167235:** Fixed an issue with [RAIL_SupportsFastRxRx\(\)](#) where it used to return an incorrect value on supported platforms.

2.15.0

- **757123:** Added support for PHY-specific RSSI offsets on the EFR32xG27 and EFR32xG28 platforms.
- **774988:** Added support for a new [RX_DIRECT_SYNCHRONOUS_MODE_DATA](#) RAIL RX data source to capture direct mode data in sync with the configured bit rate for the PHY. This requires a PHY that supports this mode of capture from the Radio Calculator and is only currently supported on the EFR32xG23.
- **824355:** Fixed an issue in IEEE802.15.4 MAC address filtering when receiving small OFDM packets.
- **832743:** Clarified use of [RAIL_SetNextTxRepeat\(\)](#) must be prior to initiating a transmit operation via API call and fixed an issue where it did not properly return an error when called while a transmit operation was in progress.
- **1032820:** Updated Packet Trace on the EFR32xG25 and EFR32xG28 when using the WiSUN protocol to have a more informative PHY identifier and to support the whole channel number range.
- **1055824:** Fixed an issue with low-side synth injection (negative IF) on proprietary 2.4GHz PHYs when using EFR32xG22 and newer chips. This fix requires regenerating the PHY with the latest version of the Radio Configurator to work.
- **1058480:** Fixed an RX FIFO corruption on EFR32xG25 that occurred when receiving/sending certain OFDM packets using FIFO mode.
- **1060146:** Fixed the RAIL_PA_CURVES_2P4_LP power curves on the EFR32xG24 to better match characterization data.
- **1061665:** Added new [RAIL_EnableCacheSynthCal](#) function to enable the radio sequencer to cache calibration values instead of recalculating them on every RX and TX event. This allows you to lower the minimum transition time for most [RAIL_StateTiming_t](#) transitions in typical cases.
- **1068856:** Corrected the sign of the frequency error reported by [RAIL_GetRxFreqOffset\(\)](#) when using OFDM on the EFR32xG25 to match how this was handled for other modulations (e.g., $\text{Freq\_error} = \text{current\_freq} - \text{expected\_freq}$).
- **1077623:** Added new [RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE](#) to trigger [RAIL_EVENT_ZWAVE_BEAM](#) on all beam frames.
- **1077623:** Added [RAIL_ZWAVE_GetBeamHomelHash\(\)](#) to retrieve the beam frame's HomelHash when handling that event and made sure that the HomelHash byte is now present on PTI for Z-Wave beam frames even when Nodeld does not match.
- **1077623:** Fixed an issue on EFR32xG23 where multiple beam frames were lumped together on PTI as one large beam chain.
- **1081396:** Adjusted channel power restrictions for the 802.15.4 PHYs on new xGM210 modules.
- **1082274:** Fixed an issue on the EFR32xG22, EFR32xG23, EFR32xG24, and EFR32xG25 chips that could cause the chip to lock up if the application attempts to re-enter EM2 within ~10 us after wake-up and hits a <0.5 us timing window. If hit, this lockup requires a power-on reset to restore normal operation to the chip.
- **1083615:** Fixed an issue for certain ramp time and power level combinations on the EFR32xG21 where the PA ramp would stop one power level short of the desired output level.
- **1086162:** Added a new [RAIL_RX_OPTION_FAST_RX2RX](#) which will force the radio sequencer to immediately transition to RXSEARCH to get ready to receive the next packet while still processing the previous one. This will minimize the RX to RX state transition time. This is only supported on chips that have [RAIL_SUPPORTS_FAST_RX2RX](#) set to true.
- **1090336:** Fixed an issue in the [Protocol Utility](#) component where BLE would be required to select a Zigbee PHY.
- **1090512:** Fixed an issue in the [Power Amplifier \(PA\) Utility](#) component where certain functions would attempt to use the RAIL_TX_POWER_MODE_2P4GIG_HIGHEST macro even though they didn't support it. This would result in undefined behavior previously, but will now correctly error.
- **1090728:** Fixed a possible RAIL_ASSERT_FAILED_UNEXPECTED_STATE_RX_FIFO issue on EFR32xG12 with [RAIL_IEEE802154_G_OPTION_GB868](#) enabled for a FEC-capable PHY which can happen when aborting a packet at frame detection, for instance by idling the radio.
- **1092769:** Fixed an issue when using Dynamic Multiprotocol and BLE Coded PHYs where a transmit could underflow depending on what protocol was active when the PHY and syncword were loaded.
- **1096663:** Fixed a compilation error in [Coexistence Utility](#) when the Coexistence WiFi TX GPIO is enabled.
- **1096665:** Fixed a compilation error in [Coexistence Utility](#) when the SL_RAIL_UTIL_COEX_WIFI_TX_PORT is defined.
- **1097152:** Added [RAIL_PacketTimeStamp_t::packetDurationUs](#) field which is currently set only on EFR32xG25 for received OFDM packets.
- **1097696:** Added RAIL support for the MGM240L lighting modules.
- **1099794:** Added separate curves when the 20 dBm PA is used at 3.3V and 1.8V for EFR32xG24.
- **1099829:** Added the new [RAIL_WMBUS_Config](#) API to allow configuring WMBUS and simultaneous M2O RX of T and C mode packets.
- **1103587:** Added a new API [RAIL_SetTxFifoAlt\(\)](#) which provides a new start offset parameter to specify where the data begins in the TX FIFO.
- **1103966:** Fixed an unexpected Rx packet abort on the EFR32xG25 when using the Wi-SUN OFDM option4 MCS0 PHY.

- **1104033:** Fixed an issue in the [RAIL_ZWAVE_ReceiveBeam](#) function so that it idles the radio regardless of whether a beam is detected on the EFR32ZG23.
- **1104441:** Fixed an issue with the [Coexistence Utility](#) counters for Zigbee that could prevent them from ticking as expected depending on how things are linked.
- **1105134:** Fixed an issue when switching between certain PHYs that could cause the first received packet to be reported as [RAIL_RX_PACKET_READY_CRC_ERROR](#) instead of [RAIL_RX_PACKET_READY_SUCCESS](#). This issue could potentially impact EFR32xG22 and newer chips.
- **1105529:** Fixed an issue on EFR32xG22 and later platforms when using a FrameType decoding PHY where a bad frame type packet was mis-reported as [RAIL_RX_PACKET_ABORT_ABORTED](#) instead of the proper [RAIL_RX_PACKET_ABORT_FORMAT](#).
- **1107288:** Added support for [RAIL_IEEE802154_SupportsRxChannelSwitching](#) on the EFR32xG21. This is also still supported at an alpha quality level on the EFR32xG24.
- **1109574:** Fixed an issue on EFR32xG22 and newer chips where a radio sequencer assert could cause the application to hang in an ISR rather than report the assert via [RAILCb_AssertFailed\(\)](#).
- **1117127:** Added [RAIL_IEEE802154_SetRxToEnhAckTx\(\)](#) to allow IEEE 802.15.4 stacks to specify a different rxToTx state transition turnaround time for Enhanced ACKs, which generally need more time to construct and secure. Immediate ACKs will continue to use the rxToTx time specified in [RAIL_IEEE802154_Config_t::timings](#).
- **1117595:** Added Tx packet duration information for EFR32xG22 and newer chips.
- **1118063:** Fixed issue with recent [RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE](#) on EFR32xG13 and EFR32xG14 where the NodeId of the promiscuous beam was not properly recorded for [RAIL_ZWAVE_GetBeamNodeId\(\)](#), causing it to report 0xFF.
- **1121630:** Increased EFR32ZG13 and EFR32ZG14 Z-Wave long-range beam detect time to improve FLIRS performance.
- **1126343:** Fixed an issue on EFR32xG24 when using the IEEE 802.15.4 PHY where the radio could become stuck when doing an LBT transmit if a frame is received during the CCA check window.
- **1134223:** Fixed an issue when using [Coexistence Utility](#) where the request line is left asserted after TX is aborted following a sync detect.
- **1135418:** Fixed incorrect [RAIL_RxPacketInfo_t::filterMask](#) on received 802.15.4 Beacon frames, which now reflects which PanId and address the Beacon's Source PanId and Source Address match, if any. Note that RAIL generally accepts all Beacons so the filterMask can be 0x00.
- **1138522:** Fixed an issue on the EFR32xG25 for SUN FSK PHYs where receiving a packet after calling [RAIL_IEEE802154_Init\(\)](#) but before configuring [RAIL_IEEE802154_ConfigGOptions](#) could break reception.
- **1140569:** Fixed a rare timing issue on EFR32xG24 where an ACK timeout might cause the next packet to be received as [RAIL_RX_PACKET_READY_CRC_ERROR](#) instead of [RAIL_RX_PACKET_READY_SUCCESS](#).
- **1150779:** Fixed the 15.4 channel configurations on the MGM240PA32 and MGM240PB32 modules to use the correct frequency for channel 26.

2.14.3

- **1058480:** Fixed an RX FIFO corruption on EFR32xG25 that occurred when receiving/sending certain OFDM packets using FIFO mode.
- **1109993:** Fix an issue in the [Coexistence Utility](#) component so that it simultaneously asserts request and priority if request and priority share the same GPIO port and polarity.
- **1118063:** Fixed issue with recent [RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE](#) on EFR32xG13 and xG14 where the NodeId of the promiscuous beam was not properly recorded for [RAIL_ZWAVE_GetBeamNodeId\(\)](#), causing it to report 0xFF.
- **1126343:** Fixed an issue on EFR32xG24 when using the IEEE 802.15.4 PHY where the radio could become stuck when doing an LBT transmit if a frame is received during the CCA check window.

2.14.2

- **747041:** Fixed an issue on the EFR32xG23 and EFR32xG25 that could cause certain radio actions to delay for extended periods of time when the main core enters EM2 while the radio is still running.
- **1077623:** Added new [RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE](#) to trigger [RAIL_EVENT_ZWAVE_BEAM](#) on all beam frames.
- **1077623:** Added [RAIL_ZWAVE_GetBeamHomelHash\(\)](#) to retrieve the beam frame's HomelHash when handling that event and made sure that the HomelHash byte is now present on PTI for Z-Wave beam frames even when NodeId does not match.
- **1077623:** Fixed an issue on EFR32ZG23 where multiple beam frames were lumped together on PTI as one large beam chain.

- **1090512:** Fixed an issue in the [Power Amplifier \(PA\) Utility](#) component where certain functions would attempt to use the `RAIL_TX_POWER_MODE_2P4GIG_HIGHEST` macro even though they didn't support it. This would result in undefined behavior previously, but will now correctly error.
- **1090728:** Fixed a possible `RAIL_ASSERT_FAILED_UNEXPECTED_STATE_RX_FIFO` issue on EFR32xG12 with `RAIL_IEEE802154_G_OPTION_GB868` enabled for a FEC- capable PHY which can happen when aborting a packet at frame detection, for instance by idling the radio.
- **1092769:** Fixed an issue when using Dynamic Multiprotocol and BLE Coded PHYs where a transmit could underflow depending on what protocol was active when the PHY and syncword were loaded.
- **1097152:** Added `RAIL_PacketTimeStamp.t::packetDurationUs` field which is currently set only on EFR32xG25 for received OFDM packets.
- **1103966:** Fixed an unexpected Rx packet abort on the EFR32xG25 when using the Wi-SUN OFDM option4 MCS0 PHY.
- **1105134:** Fixed an issue when switching certain PHYs that could cause the first received packet to be reported as `RAIL_RX_PACKET_READY_CRC_ERROR` instead of `RAIL_RX_PACKET_READY_SUCCESS`. This issue could potentially impact EFR32xG22 and newer chips.
- **1109574:** Fixed an issue on EFR32xG22 and newer chips where a radio sequencer assert could cause the application to hang in an ISR rather than report the assert via `RAILCb_AssertFailed()`.

2.14.1

- **1068856:** Corrected the sign of the frequency error reported by `RAIL_GetRxFreqOffset()` when using OFDM on the EFR32xG25 to match how this was handled for other modulations (e.g. `Freq_error=current_freq- expected_freq`).
- **1074384:** The `RAIL_SetTune()` and `RAIL_GetTune()` functions now use the `CMU_HFXOCTuneSet()` and `CMU_HFXOCTuneGet()` functions respectively on EFR32xG2x and newer devices.
- **1077611:** Fixed an issue on the EFR32xG25 that would cause a 40us porch before an OFDM TX.
- **1082274:** Fixed an issue on the EFR32xG22, EFR32xG23, EFR32xG24, and EFR32xG25 chips that could cause the chip to lock up if the application attempts to re-enter EM2 within ~10 us after wake-up and hits a <0.5 us timing window. If hit, this lockup requires a power on reset to restore normal operation to the chip.

2.14.0

- **670716:** The `RAIL_ConfigRfSenseSelectiveOokWakeupPhy()` will now return an error when run on the EFR32xG21 platform because this device cannot support the wakeup PHY.
- **751910:** Added HFXO temperature compensation in RAIL on platforms that support `RAIL_SUPPORTS_HFXO_COMPENSATION`. This feature can be configured with the new `RAIL_ConfigHFXOCompensation()` API. The user will also need to be sure to handle the new `RAIL_EVENT_THERMISTOR_DONE` event to trigger a call to `RAIL_CalibrateHFXO` to perform the compensation.
- **765112:** Added options in [Protocol Utility](#) to control whether Z-Wave, 802.15.4 2.4 GHz and Sub-GHz, and BLE are enabled so that the user can save space in their application by disabling unused protocols.
- **843708:** Moved function declarations from `rail_features.h` to `rail.h` to avoid a convoluted include dependency order.
- **844325:** Fixed `RAIL_SetTxFifo()` to properly return 0 (error) rather than 4096 for an undersized FIFO.
- **845608:** Fixed an issue with the `RAIL_ConfigSyncWords` API when using certain underlying demodulator hardware on EFR32xG2x parts.
- **846240:** Added a new API `RAIL_ZWAVE_PerformIrcal` to help perform an IR calibration across all the different PHYs used by a Z-Wave device.
- **851150:** Fixed an issue on EFR32xG2 series devices where the radio would trigger `RAIL_ASSERT_SEQUENCER_FAULT` when PTI is used and GPIO configuration is locked. GPIO configuration can only be locked when PTI is disabled. See `RAIL_EnablePti()` for further information.
- **853682:** Added 40MHz crystal support on EFR32xG24 devices to the [Built-in PHYs for alternate HFXO frequencies](#) component.
- **855048:** Updated the `pa_customer_curve_fits.py` helper script to accept floating point value for the maximum power argument, similar to the increment argument.
- **857210:** Added support in [Coexistence Utility](#) for configuring priority options when directional priority is enabled but no static priority GPIO is defined.
- **857267:** Fixed an issue when using the [Coexistence Utility](#) component with TX abort, the signal identifier feature and DMP.
- **858567:** Added support for IEEE 802.15.4 fast RX channel switching with the new `RAIL_IEEE802154_ConfigRxChannelSwitching` API on supported platforms (`RAIL_IEEE802154_SupportsRxChannelSwitching`).

This feature allows us to simultaneously detect packets on any two 2.4GHz 802.15.4 channels with a slight reduction in overall sensitivity of the PHY.

- **988492:** Added support for the BGM220SC23HNA2 module.
- **1015152:** Fixed an issue on EFR32xG2x devices where [RAIL_EVENT_RX_FIFO_ALMOST_FULL](#) or [RAIL_EVENT_TX_FIFO_ALMOST_EMPTY](#) could trigger improperly when the event is enabled or the FIFO is reset.
- **1017609:** Fixed an issue where PTI appended information could be corrupted when [RAIL_RX_OPTION_TRACK_ABORTED_FRAMES](#) is in effect when [RAIL_IDLE_FORCE_SHUTDOWN](#) or [RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS](#) is used. Also clarified that [RAIL_RX_OPTION_TRACK_ABORTED_FRAMES](#) is not useful with coded PHYs.
- **1019590:** Fixed an issue when using the [Coexistence Utility](#) component with BLE where the `sl_bt_system_get_counters()` function would always return 0 for GRANT denied counts.
- **1019794:** Eliminated compiler warning in [Initialization Utility](#) component when few of its features are enabled.
- **1022182:** Added a new [Thermal Protection](#) feature, on platforms that support [RAIL_SUPPORTS_THERMAL_PROTECTION](#), to track temperature and prevent transmits when the chip is too hot.
- **1022692:** Added new table based OFDM and FSK PAs for EFR32xG25 based devices. The output power of these can be modified through a new customer provided look-up table. Ask support or look for an updated app note on how to configure the values in this table for your board.
- **1023016:** Fixed an issue on EFR32xG22 and newer chips where waits in between radio activity would consume slightly more power than necessary after the first 13ms. This was especially noticeable when using [RAIL_ConfigRxDutyCycle](#) with large off time values.
- **1024294:** Removed support for the deprecated BGX220P22HNA and BGX220S22HNA modules.
- **1024457:** Added support for the MGM240SA22VNA, BGM240SA22VNA, and BGM241SD22VNA modules and updated the configurations for the BGM240SB22VNA, MGM240SB22VNA, and the MGM240SD22VNA.
- **1029740:** Fixed issue where [RAIL_GetRssi\(\)](#)/[RAIL_GetRssiAlt\(\)](#) could return a "stale" RSSI value (the value was from previous RX state instead of the current one) if called quickly upon entering receive.
- **1040814:** Added support to the [Coexistence Utility](#) component for configuring the coexistence request priority on sync detect when using BLE.
- **1041734:** Broke up some EFR32xG12 802.15.4 dynamic FEC code to save code size for Zigbee and BLE which never need this functionality.
- **1056207:** Fixed an issue with IQ sampling when using the [Angle of Arrival/Departure \(AoX\) Utility](#) component with only 0 or 1 antennas selected.
- **1057830:** Remove [Coexistence Utility](#) dependency from the RAIL Utility, Coulomb Counter component.
- **1062712:** Fixed an issue where the [Coexistence Utility](#) component wouldn't always update request states correctly which could lead to missed events triggered by new requests.
- **1062940:** Prevent the [Coexistence Utility](#) component from aborting BLE transmits when `SL_RAIL_UTIL_COEX_BLE_TX_ABORT` is disabled.
- **1063152:** Fixed an issue where radio reception would not be fully cleaned up when a receive error occurs with receive state transitions set to idle on error but transmit on success, a configuration mostly associated with BLE. On the EFR32xG24 this could cause a SYNTH calibration to not be properly restored and eventually cause the radio to stop working.
- **1063502:** The [RAIL_PrepareChannel\(\)](#) function has been made dynamic multi-protocol safe and will no longer return an error if called when your protocol is inactive.

2.13.3

- **1041997:** Fixed the librail_config libraries for the following xGM240 modules: BGM240PA22VNA, BGM240PA32VNA, BGM240PA32VNN, BGM240PB22VNA, BGM240PB32VNA, BGM240PB32VNN, MGM240PA22VNA, MGM240PA32VNA, MGM240PA32VNN. Without this update these modules will assert when trying to load the supported BLE and IEEE 802.15.4 PHYs.

2.13.2

- **813171:** Added early support for IEEE802.15.4G dynamic forward error correction PHYs on the EFR32xG12 platform. Using this at this time will require help from support.
- **844377:** Fixed a BLE 2Mbps AoX issue on EFR32xG24 when using a 38.4MHz crystal.
- **1026914:** Improved PA configurations for the xGM240 modules based on additional test data.
- **1029710:** Fixed an issue with RAIL's PA auto mode that would cause us to select an unsupported [RAIL_TxPowerMode_t](#) on chip OPNs that are missing the higher power PAs.

2.13.1

- **819544:** Improved reception on EFR32xG23 and EFR32xG24 when using a PHY with fast detect enable (preamble sense mode).
- **843708:** Moved function declarations from [rail_features.h](#) to [rail.h](#) to avoid a convoluted include dependency order.
- **844325:** Fixed [RAIL_SetTxFifo\(\)](#) to properly return 0 (error) rather than 4096 for an undersized FIFO.
- **844936:** Fixed an issue where using [RAIL_SetNextTxRepeat\(\)](#) could cause a brownout reset on EFR32xG23.
- **853714:** Fixed an issue with xGM240 modules causing them to assert during initialization.
- **857210:** Added support in [Coexistence Utility](#) for configuring priority options when directional priority is enabled but no static priority GPIO is defined.
- **988518:** Fixed an issue where the radio sequencer would leave portions of the chip enabled after AoX CTE packet reception, preventing the device from going into EM2 sleep and potentially causing missed packet receive events.

2.13.0

- **376658:** Fixed an issue with the BLE coded PHY on EFR32xG21 where a packet received with a corrupt coding indicator would result in an invalid start of packet timestamp.
- **645362:** The [Packet Trace Interface \(PTI\) Utility](#) component will now validate that the correct set of pins are in use for the desired PTI mode.
- **653200:** RAIL will now error if attempting to start a CSMA or LBT transmit while a scheduled RX is still in progress or vice versa.
- **714958:** The RAIL channel of a received packet is now available in the packet's [RAIL_RxPacketDetails_t::channel](#) field. This can be of value when scanning or hopping across multiple channels while letting packets accumulate in the receive FIFO for later processing.
- **739371:** Added the [RAIL_ConfigPaAutoEntry](#) API to allow for easier configuration of PA auto mode operation in RAIL.
- **746775:** Added PA curves for BGM240P and MGM240P modules.
- **748680:** Added the [RAIL_SetRssiDetectThreshold](#) API to allow the user to detect when the RSSI is at or above a configurable threshold. Once configured the [RAIL_EVENT_DETECT_RSSI_THRESHOLD](#) event can be used to detect when this happens.
- **758341:** Restricted the [SL_RAIL_UTIL_PA_RAMP_TIME_US](#) to 10us on some EFR32 modules to match the certification conditions.
- **772769:** Fixed an issue when running IR Calibration on the EFR32xG23 using [RAIL_CalibrateIrAit](#) where we could compute an invalid IRCAL value for certain PHYs and chips.
- **777427:** Fixed support for using the signal identifier CCA modes simultaneously with a user-enabled signal identifier trigger event.
- **812815:** Added support for the MGM240L022RNF module.
- **813381:** Added support for the FGM230SA27HGN and FGM230SBHGN modules.
- **819644:** Fixed an issue with frame-type decoding PHYs running at more than 500kbps on EFR32xG22 and later.
- **825083:** Fixed an issue on EFR32xG23 and EFR32xG24 where PTI could merge multiple receive packets into the same transaction when interrupt latency is significant.
- **828716:** Added the [RAIL_GetChannelAlt](#) API. This function returns the channel the radio is currently using. If using DMP and run on the inactive protocol it returns the channel that will be used when next switching to that protocol. When using channel hopping, mode switch, and other features that change channels dynamically this may be different than what is returned by [RAIL_GetChannel](#) as this function will track what channel the radio is actually on at that moment and not what it started on.
- **829499:** Fixed an issue where [RAIL_GetRadioStateDetail](#) would not report the correct state information when frame detection was disabled or during an LBT operation.
- **830214:** Ensure that the [RAIL_RadioConfigChangedCallback_t](#) is called for all RAIL handles in a dynamic multi-protocol application where multiple handles use the same underlying PHY configuration.
- **835299:** Fixed an issue with dynamic handling of whitening and FCS in FSK when only [RAIL_IEEE802154_E_OPTION_GB868](#) was enabled.
- **844600:** Fixed issue of not being able to receive packets during a [RAIL_ScheduleRx](#) configured with a zero relative start time when Power Manager sleep is enabled and configured with an EM2 or lower energy requirement.

2.12.2

- **759793:** Fixed an issue with BLE long-range reception on EFR32xG21 that corrupted packet data and tripped [RAIL_ASSERT_FAILED_UNEXPECTED_STATE_RX_FIFO](#).
- **764346:** BLE and 802.15.4 built-in PHYs are now exposed as public symbols in header files. Do not modify these without explicit instructions by Silicon Labs.
- **764347:** Added a new [Built-in PHYs for alternate HFXO frequencies](#) component to enable the built-in phys to operate with either 38.4 MHz or 39 MHz crystals on EFR32xG24 devices.
- **774883:** Updated power curves for ZGM230SA27HGN, ZGM230SA27HNN, ZGM230SB27HGN modules to provide more accurate output powers at the lower and higher end of the dBm range.
- **777290:** The PA auto mode configuration is fixed to use both HP and LP PA modes on the 10dBm EFR32xG24 chips.
- **777427:** Fixed support for using the signal identifier CCA modes simultaneously with a user enabled signal identifier trigger event.
- **812938:** Fixed the [RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE](#) mode on EFR32xG22 and EFR32xG23 to properly stay active after detecting Timing and Preamble when using the standard demodulator.
- **813182:** Previously selecting an invalid CCA mode using [RAIL_IEEE802154_ConfigCcaMode](#) would fail silently and continue to use the RSSI based CCA mode. Now [RAIL_IEEE802154_ConfigCcaMode](#) will return an error if an invalid CCA mode is selected.

2.12.1

- **758341:** Restricted the [SL_RAIL_UTIL_PA_RAMP_TIME_US](#) to 10us on some EFR32 modules to match the certification conditions.
- **764234:** Changed Quuppa channel 2 frequency from 2480 MHz to 2403 MHz.
- **765843:** Updated the Z-Wave PHYs for the EFR32xG23 to prevent a sensitivity degradation on the R2 (9.6 kbps) PHY.
- **773178:** Fixed compiler warning in [Callbacks Utility](#) component when app_assert component ignores asserts.

2.12.0

- **646980:** An attempt to use an unsupported built-in radio channel configuration, e.g. on a module that does not support that protocol or configuration, will now trip [RAIL_ASSERT_FAILED_INVALID_CHANNEL_CONFIG](#) rather than returning success and ignoring the configuration.
- **671651:** Fixed timing problems with certain [State Transitions](#) or [RX Channel Hopping](#) delay values on the EFR32xG22 and newer parts.
- **682739:** Fixed an issue with the BLE coded PHY's modulation index on the EFR32xG21 parts that could cause deviation measurements to fail.
- **688211:** Added IEEE802.15.4 Coexistence and FEM PHYs to EFR32xG12 and EFR32xG13 based modules.
- **688211:** Updated IEEE802.15.4 FEM PHYs on EFR32xG12 and EFR32xG13 based modules for improved performance.
- **697705:** Added support for the ZGM230SA27HGN, ZGM230SA27HNN, and ZGM230SB27HGN modules.
- **708206:** Added [RAIL_GetTxPacketsRemaining\(\)](#) API for use when handling one of the [RAIL_EVENTS_TX_COMPLETION](#) to get a sense of how many transmits remain in a [RAIL_SetNextTxRepeat\(\)](#) sequence.
- **712012:** Updated all header files to have extern "C" when being built with C++ for compatibility.
- **714271:** Fixed an issue where [RAIL_IEEE802154_Config2p4GHzRadio*\(\)](#) and [RAIL_IEEE802154_ConfigGB*Radio\(\)](#) functions were improperly clearing or setting certain [RAIL_IEEE802154_EOptions_t](#). Also documented that these functions still implicitly clear or set certain [RAIL_IEEE802154_GOptions_t](#) suitable for that configuration.
- **714709:** Made the [RAIL_EnableRxDutyCycle\(\)](#) API safe to call in a multiprotocol application.
- **715123:** Added PA curves for HP, MP, LP and LLP modes on all EFR32xG23 radio boards.
- **716369:** Fixed an issue where incorrect radio transition times were being applied at higher temperatures when using the high power PA on EFR32xG22 parts.
- **718290:** Changed the LQI measurement to use chip errors on IEEE802.15.4 PHYs for EFR32xG24 parts.
- **719194:** Added [RAIL_PA_BAND_COUNT](#) to count [RAIL_PaBand_t](#).
- **720948:** Added a new [RAIL_RxDataSource_t](#) to capture direct mode data on supported devices.
- **721163:** To save both flash and RAM, moved information formerly contained in [RAIL_Config_t::protocol](#), [RAIL_Config_t::scheduler](#), and [RAIL_Config_t::buffer](#) internal to RAIL and sized appropriately for single vs. multiprotocol. RAIL multiprotocol now provides two internal state buffers for two protocols by default. An application that needs more must now call [RAIL_AddStateBuffer3\(\)](#) or [RAIL_AddStateBuffer4\(\)](#) to add a 3rd and 4th buffer, respectively. Otherwise [RAIL_Init\(\)](#) will fail when trying to initialize a 3rd or 4th protocol.
- **723098:** Fixed [RAIL_SetFixedLength\(handle, RAIL_SETFIXEDLENGTH_INVALID\)](#) to restore dynamic frame length operation if the current PHY was originally configured for that.
- **723558:** Added support for BGM240PA22VNA, BGM240PA32VNA, BGM240PA32VNN, BGM240PB22VNA, BGM240PB32VNA, BGM240PB32VNN, MGM240PA22VNA, MGM240PA32VNA, MGM240PA32VNN, MGM240PB22VNA,

MGM240PB32VNA and MGM240PB32VNN modules.

- **723605:** [Coexistence Utility](#) GPIO interrupt numbers are now chosen at runtime to avoid conflicts.
- **726491:** A new RAIL API [RAIL_GetSchedulerStatusAlt](#) will now return more descriptive radio scheduler events as well as the [RAIL_Status_t](#) of the RAIL API invoked by the radio scheduler. As a part of the new API, new [RAIL_SchedulerStatus_t](#) events have been added while retaining the previous ones for backwards compatibility. Note that the underlying values of the existing [RAIL_SchedulerStatus_t](#) events may have changed.
- **727721:** Updated the [RAIL_IEEE802154_Config2p4GHzRadio*Fem](#) PHYs on the EFR32xG12 and EFR32xG13 devices to improve performance.
- **727728:** Added a new [RAIL_IEEE802154_Config2p4GHzRadioCustom1](#) API to configure an alternate IEEE 802.15.4 PHY with slightly different performance characteristics for the EFR32xG12 and EFR32xG13 parts. Use this API if instructed by Silicon Labs for your use case.
- **738931:** Fixed an issue with the BLE Coded PHYs on the EFR32xG22 device that could cause some packets to be improperly sent and not trigger a [RAIL_EVENT_TX_PACKET_SENT](#) event.
- **739594:** Fixed an issue with the [RX_IQDATA_FILTLSB_RAIL_RxDataSource_t](#) on EFR32xG23 parts where the data did not properly saturate and was instead just the lower 16 bits of IQ sample data.
- **744323:** Fixed an issue when using BLE AoX where non-AoX packets were transmitted on an undefined antenna. They will now always use the first antenna in the configured [RAIL_BLE_AoxConfig_t::antArrayAddr](#) pattern.
- **745528:** Fixed some incorrect [RAIL_RxPacketInfo_t::filterMask](#) values for 802.15.4 ACKs when promiscuous, or when the PanId coordinator received a packet with only source PanId and no destination address.
- **753655:** The example CSV files referred to in AN1127: Power Amplifier Power Conversion Functions in RAIL 2.x are updated with realistic values.
- **753860:** Fixed an issue when running IR Calibration on the EFR32xG23 ([RAIL_CalibrateIrAlt](#)) where we could compute a completely invalid IRCAL value for certain PHYs and chips.
- **754219:** Increased maximum BLE Coexistence request window setting, [SL_RAIL_UTIL_COEX_REQ_WINDOW](#), in [Coexistence Utility](#) from 255 to 5000.

2.11.3

- **737292:** Fixed an issue on the EFR32xG21 that was causing some BLE coded PHY packets to be sent with a corrupted sync word and payload breaking the connection.
- **738158:** Fixed the [RX_DIRECT_MODE_DATARAIL_RxDataSource_t](#) data source on the EFR32xG23.
- **741120:** Fixed an issue where the IQ data captured during BLE's CTE could be all zeros on the 2Mbps PHY.
- **743258:** Fixed an issue that could cause the antenna switching pattern in BLE AoX applications to sometimes repeat a previous sequence instead of following the expected antenna sequence.

2.11.2

- **274536:** Added a new [RAIL_ConfigDirectMode\(\)](#) API for configuring direct mode settings on all chips. This can set whether the data stream is synchronous or asynchronous and which GPIOs are used for the feature.
- **671651:** Fixed timing problems with certain [State Transitions](#) or [RX Channel Hopping](#) delay values on the EFR32xG22 and newer parts.
- **714271:** Fixed an issue where [RAIL_IEEE802154_Config2p4GHzRadio*\(\)](#) and [RAIL_IEEE802154_ConfigGB*Radio\(\)](#) functions were improperly clearing or setting certain [RAIL_IEEE802154_EOptions_t](#). Also documented that these functions still implicitly clear or set certain [RAIL_IEEE802154_GOptions_t](#) suitable for that configuration.
- **715123:** Added PA curves for EFR32xG23 for HP, MP, LP and LLP modes for both 14dBm and 20dBm variants.
- **716369:** Fixed an issue where incorrect radio transition times were being applied at higher temperatures when using the high power PA on EFR32xG22 parts.
- **720948:** Added a new [RAIL_RxDataSource_t](#) to capture direct mode data on supported devices.
- **735862:** Now when using a [RAIL_RxDataSource_t](#) other than [RX_PACKET_DATA](#) the receiver will be disabled any time a [RAIL_EVENT_RX_FIFO_OVERFLOW](#) occurs and need to be manually restarted. If the [RAIL_EVENT_RX_FIFO_OVERFLOW](#) event is not enabled then the receiver will continue to run and lose some amount of data until the buffer is processed which matches the old behavior.

2.11.1

- **646980:** An attempt to use an unsupported built-in radio channel configuration, e.g. on a module that does not support that protocol or configuration, will now trip [RAIL_ASSERT_FAILED_INVALID_CHANNEL_CONFIG](#) rather than returning success and ignoring the configuration.

- **675252:** Fixed an antenna diversity regression where a transmitted auto-ACK would incorrectly go out the currently configured TX antenna rather than the antenna used to receive the packet being acknowledged.
- **676896:** Fixed an issue in OOK PHYs where dynamic adjustments made to receive a packet with a high RSSI could persist after the packet and decrease the ability to receive packets with a lower RSSI.
- **696198:** Fixed an issue on PHYs that do not support dual sync words. If [RAIL_RX_OPTION_ENABLE_DUALSYNC](#) is called when one of these PHYs is loaded the [RAIL_ConfigRxOptions](#) function will now return [RAIL_STATUS_INVALID_PARAMETER](#).
- **697097:** Fixed a rare situation where a premature [RAIL_EVENT_TX_BLOCKED](#) event might occur when auto-ACK is enabled and a scheduled transmit using [RAIL_SCHEDULED_TX_DURING_RX_ABORT_TX](#) is pending when an erroneous packet is received.
- **700439:** Fixed an issue where configuring a Selective RF Sense Wakeup packet by calling [RAIL_ConfigRfSenseSelectiveOokWakeupPhy](#) followed by [RAIL_SetRfSenseSelectiveOokWakeupPayload](#) would put RAIL in fixed length mode and leave it there even after a new PHY was loaded. We will now revert the fixed length settings to their default when loading a new PHY after the Selective RF Sense Wakeup PHY was loaded so that the incoming PHY's settings are used.
- **710273:** Fixed an issue using [RAIL_TX_REPEAT_OPTION_HOP](#) with [RAIL_SetNextTxRepeat\(\)](#) or [RAIL_BLE_SetNextTxRepeat\(\)](#) when they return an error found in the channel-hopping configuration, yet would still attempt to repeat the next transmit anyway.
- **710983:** Added the new [RAIL_STREAM_CARRIER_WAVE_PHASENOISE](#) [RAIL_StreamMode_t](#) for phase noise measurement.
- **719194:** Added [RAIL_PA_BAND_COUNT](#) to count [RAIL_PaBand_t](#).

2.11.0

- **369712:** Added [RAIL_StartScheduledCcaCsmaTx](#) and [RAIL_StartScheduledCcaLbtTx](#) APIs to allow applications to schedule a CSMA/LBT transmit.
- **456701:** Fixed an issue on EFR32xG1x parts where calling [RAIL_Init\(\)](#) with the MSC->CTRL.CLKDISFAULTEN bit set would cause a bus fault.
- **524885:** Added support for a new [RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND](#), triggered on the reception of a Z-Wave Long range packet with acknowledgement request bit set, following which the application must call [RAIL_ZWAVE_SetLrAckData](#) to populate the fields of the Z-Wave Long range acknowledgement packet.
- **626961:** Add support for the [RAIL_TX_POWER_MODE_*_HIGHEST](#) options in the [RAIL_GetTxPowerCurve\(\)](#) function.
- **654600:** On EFR32xG21, a watchdog has been added to terminate an RSSI averaging operation in case the [RAIL_EVENT_RSSI_AVERAGE_DONE](#) event does not occur.
- **655541:** Fixed an issue on EFR32xG22 and later where packet filtering might be incomplete in FEC-enabled radio configurations causing good packets to be improperly dropped. Note that if packet filtering fails close to the end of an otherwise successfully received packet, the packet may be flagged [RAIL_RX_PACKET_READY_CRC_ERROR](#) rather than [RAIL_RX_PACKET_ABORT_FILTERED](#). This issue is still possible on earlier chips due to hardware filtering restrictions.
- **660021:** Changed when [RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#) event is issued to better facilitate support for 802.15.4E-2012 Enhanced ACKing and reduce time spent in the event handler. The event is now issued **after** receiving the Auxiliary Security Header (if present) in the MAC Header of the incoming frame, and for MAC Command frames, after receiving the MAC Command byte (which may be encrypted). This change is **not** backwards compatible with prior releases for Enhanced ACK requesting frames, but is for Immediate ACK requesting frames. Use of [RAIL_IEEE802154_EnableEarlyFramePending\(\)](#) is no longer required to support Enhanced ACKing; the notion of early frame pending notification has also shifted to after the Auxiliary Security Header for Enhanced ACK requesting frames.
- **661651:** Changed RAIL Timer Synchronization over sleep on the EFR32xG21 to use the RTCC instead of the PRORTC to reduce current consumption in EM2.
- **665705:** Fixed an issue where a transmit with [RAIL_TX_OPTION_SYNC_WORD_ID](#) 1 to use SYNC2 would improperly indicate SYNC1 was used in the packet trace appended information.
- **666275:** Fixed potential delays when using RAIL's channel hopping or duty cycling features in EM1P mode on the EFR32xG22 and newer parts.
- **667103:** Fixed [RAIL_ReadRxFifo\(\)](#) to behave as documented when passed NULL dataPtr: the data is thrown away rather than copied out.
- **671817:** Fixed an issue when switching between certain radio configurations (e.g. Z-Wave) where use of [RAIL_TX_OPTION_REMOVE_CRC](#) can become permanently stuck.
- **673333:** Fixed an issue with [RAIL_TX_OPTION_WAIT_FOR_ACK](#) transmits where an RX overflow during the ACK wait period would silently abort the ACK timer resulting in no [RAIL_EVENT_RX_ACK_TIMEOUT](#) being generated.

- **681133:** Added [RAIL_TX_OPTION_RESEND](#) to allow re-transmitting the packet most recently loaded into the Transmit FIFO. This can be used in combination with [RAIL_SetNextTxRepeat\(\)](#) on supported platforms to repeatedly transmit the same packet.
- **681135:** Added new [RAIL_SetNextTxRepeat\(\)](#) API and [RAIL_TxRepeatConfig_t](#) to allow configuration of repeated transmits triggered by an initial transmit.
- **681136:** Added ability to configure TxToTx state transition time via [RAIL_StateTiming_t::txToTx](#) and [RAIL_SetStateTiming\(\)](#). This time is generally used between an autoACK transmit and a user transmit that was pending. It's also used during repeated transmits by default, but can be overridden by the [RAIL_TxRepeatConfig_t::delayOrHop](#) or [RAIL_BLE_TxRepeatConfig_t::delayOrHop](#) configuration.
- **681141:** Added ability to hop channels during repeated transmits in [RAIL_TxRepeatConfig_t](#) using [RAIL_TX_REPEAT_OPTION_HOP](#).
- **681143:** Added BLE-specific hooks for repeated transmits with channel hopping in [RAIL_BLE_SetNextTxRepeat\(\)](#) and [RAIL_BLE_TxRepeatConfig_t](#).
- **682032:** Fix an issue where setting [RAIL_SCHEDULED_TX_DURING_RX_ABORT_TX](#) for a scheduled transmit could cause subsequent non-scheduled transmits to be blocked.
- **695484:** The posting of [RAIL_EVENT_TX_ABORTED](#) now occurs before the PA ramps down, at the same time other transmit completion events get posted.
- **696665:** Fixed an issue in [RAIL_ConvertRawToDbm](#) for PAs which use piecewise-linear line segment fit, where the minimum raw power level was incorrectly compared to the minimum deci-dBm value.
- **697097:** Fixed a rare situation where a premature [RAIL_EVENT_TX_BLOCKED](#) event might occur when auto-ACK is enabled and a scheduled transmit using [RAIL_SCHEDULED_TX_DURING_RX_ABORT_TX](#) is pending when an erroneous packet is received.
- **699890:** Fixed missing C++ compatibility in the [Initialization Utility](#) component's generated header file.
- **701604:** Fixed an issue where using [RX Channel Hopping](#) with channels in different frequency bands would cause RAIL to assert.
- **703788:** Fixed an issue on EFR32xG2x devices where RAIL would not allow you to initialize the radio with voltage scaling enabled even though this is supported on these devices.
- **705595:** Fixed an issue where custom PA curves provided through the PA Module were not respected when building Silicon Labs Zigbee applications.
- **705802:** Allow [RAIL_ZWAVE_ReceiveBeam](#) to be run on US LR regions instead of always returning an error.
- **706542:** The [Callbacks Utility](#) component can now be configured to not provide the [RAILCb_AssertFailed\(\)](#) function for situations where the application wants to provide its own implementation.
- **708511:** Fixed possible [RAIL_ASSERT_FAILED_UNEXPECTED_STATE_TX_FIFO](#) when [RAIL_SetTxFifo\(\)](#) is called after the transmit FIFO had been filled enough to cause its write offset to wrap.

2.10.2

- **652028:** If the selected [RAIL_TxPower_t](#) is not supported by the chip, then by default [RAIL_TX_POWER_MODE_2P4_HIGHEST](#) is chosen as the power mode.
- **666275:** Fixed potential delays when using RAIL's channel hopping or duty cycling features in EM1P mode on the EFR32xG22 and newer parts.
- **666917:** Previously runtime configuring the Zigbee coexistence PHY select timeout to 255 ms would only guarantee 255 ms on the coexistence optimized PHY. Now the coexistence optimized PHY will be selected indefinitely after runtime configuring the coexistence PHY select timeout to 255 ms.
- **667555:** Fixed an issue with the [Front End Module \(FEM\) Utility](#) component where it would generate an error if the Rx pin was not required and turned off by the user.
- **669697:** Updated the configuration libraries on BGM210PB22JIA and BGM210PB32JIA modules to fix issues using these parts in the field.
- **671817:** Fixed an issue when switching between certain radio configurations (e.g. Z-Wave) where use of [RAIL_TX_OPTION_REMOVE_CRC](#) can become permanently stuck.
- **672904:** The minimum and maximum power levels for BGM220P and BGM220SC modules are updated to be 0 and 15 respectively.
- **672909:** Fixed an issue with PA auto mode where it might try to use a [RAIL_TxPowerMode_t](#) that is not valid for the current part.
- **673246:** A multiprotocol scheduled receive with a relative start time and an absolute end time now respects that end time regardless of when the receive actually starts. Before, the end time was made relative and pushed out based on the actual

start time to keep the receive window the same width in duration. This is a backwards-incompatible change, but should align better with what users expected when specifying an absolute end time.

- **684407:** Fixed an issue where calling [RAIL_ConfigAntenna\(\)](#) on EFR32xG22 and newer devices could fault if the GPIO block clock was not explicitly enabled beforehand.
- **687455:** Fixed the conversion of power level to dBm to fetch correct values for BGM220P and BGM220S modules when using the RAIL_TX_POWER_MODE_2P4GIG_HP power mode and power levels above the maximum.

2.10.1

- **635037:** Added support on the EFR32xG22 parts for a new BLE PHY that can receive 1Mbps and LR Coded packets simultaneously. See [RAIL_BLE_ConfigPhySimulscan\(\)](#).
- **652969:** Restored automatic IR calibration on EFR32xG22 at [RAIL_Init\(\)](#) time.
- **653955:** Fixed an issue when using the [Coexistence Utility](#) component where PWM was enabled if SL_RAIL_UTIL_COEX_PWM_DEFAULT_ENABLED was set, even if SL_RAIL_UTIL_COEX_PWM_REQ_ENABLED was disabled. Now SL_RAIL_UTIL_COEX_PWM_DEFAULT_ENABLED is ignored when SL_RAIL_UTIL_COEX_PWM_REQ_ENABLED is disabled.
- **654726:** Fixed an issue where antenna diversity was not being enabled on EFR32xG1x devices when configured through the [Antenna Diversity Utility](#) component.
- **656175:** Fixed an issue where any [RAIL_ChannelConfigEntry_t](#) in a [RAIL_ChannelConfig_t](#) with a maximum power less than 0 dBm would be stuck at the maximum power no matter what power was requested.
- **663815:** Fixed an issue when using the [Antenna Diversity Utility](#) component where moving from TX antenna mode HAL_ANTENNA_MODE_ENABLE2 to HAL_ANTENNA_MODE_DIVERSITY would cause the antenna selection to get stuck on antenna 2.
- **665161:** Fixed an issue on EFR32xG22 and later PTI where the network analyzer could misrepresent the RSSI of incoming packets.

2.10.0

- **244222:** Added support for reporting more detailed transmit errors on the Packet Trace Interface (PTI).
- **318768:** Added new API [RAIL_GetRadioStateDetail](#) that provides more detailed radio state information than [RAIL_GetRadioState](#).
- **362133:** The default RSSI offset on the EFR32xG1, EFR32xG12, EFR32xG13, EFR32xG14, and EFR32xG21 chips does not compensate for a known internal hardware offset. This offset is chip specific and can be found using the new [Received Signal Strength Indicator \(RSSI\) Utility](#) component which will load the correct value for your chip by default when the RSSI Offset component or Hardware Configurator peripheral is enabled. Since the hardware and antenna design can also impact this offset it is recommended that you measure this value for your particular hardware for the best accuracy. This correction is not enabled by default on the chips listed above to prevent changing radio behavior significantly without the user opting into this change. For the EFR32xG22 and future chips the hardware offset is measured and included by default. Please be aware that if any prior CCA token modifications have been made to modify the threshold for CSMA/LBT transmits, adding an RSSI Offset value could reduce or prevent TX.
- **446308:** Updated [RAIL_ZWAVE_ReceiveBeam\(\)](#) to automatically idle the radio when [RAIL_ZWAVE_ReceiveBeam\(\)](#) finishes even when no beam is detected.
- **451099:** Added the ability to use the [Initialization Utility](#) component multiple times when creating a multiprotocol application.
- **471715:** Fixed an issue when using [RAIL_ConfigAntenna\(\)](#) on the EFR32xG22 with an RF path other than 0 since these parts do not have multiple RF paths.
- **478948:** Added [RAIL_RxPacketInfo_t::filterMask](#) field of type [RAIL_AddrFilterMask_t](#) which is a bitmask representing which address filter(s) the packet has passed.
- **493557:** Added the ability for [RAIL_GetRssi\(\)](#) to wait for a valid RSSI in radio states that are transitioning into RX. Additionally, a maximum wait timeout for a valid RSSI can be configured using the new API [RAIL_GetRssiAlt\(\)](#).
- **494571:** Added a new [RAIL_ZWAVE_ConfigRxChannelHopping\(\)](#) API to configure Z-Wave Rx channel hopping using the recommended hopping parameters.
- **494571:** Added a new [RAIL_ZWAVE_GetRegion\(\)](#) API to determine the currently selected Z-Wave region.
- **495497:** Changed [RAIL_PacketTimeStamp_t::totalPacketBytes](#) from uint32_t to uint16_t to reduce RAM usage.
- **497196:** The [Initialization Utility](#) component now defaults most options to a disabled state, instead of enabled. Now you have to opt-in, instead of opt-out, of RAIL init functionality.
- **501510:** Added a new [RAIL_SupportsTxPowerModeAlt\(\)](#) API to get the minimum and maximum power levels for a specific power mode if the power mode is supported by the chip.

- **519195:** The EFR32xG21 will now use RTCC channel 0, as opposed to the PRORTC, to perform sleep timer synchronization. This will help lower the EM2 current consumption for this chip.
- **519288:** The [Power Amplifier \(PA\) Utility](#) component now enables PA calibration by default to ensure that PA power remains consistent chip-to-chip.
- **520521:** Added a new API [RAIL_SetAddressFilterAddressMask\(\)](#) that allows for setting a bit mask pattern for packet data in the address filters.
- **520937:** Add new [RAIL_EVENT_RF_SENSED](#) as an alternative to the current [RAIL_StartRfSense\(\)](#) callback parameter.
- **520943:** Added a new API [RAIL_ConfigSleepAlt\(\)](#) to allow configuring the PRS channel, RTCC channel, and whether sleep is enabled in one call.
- **524046:** Added support for MGM210PB22JIA, MGM210PB32JIA, BGM210PB22JIA and BGM210PA32JIA modules.
- **580312:** Created a new [Protocol Utility](#) component for setting up RAIL to use one of the standards based PHYs by default.
- **630457:** On custom boards, the [Packet Trace Interface \(PTI\) Utility](#) component no longer reserves pins for use without being configured.
- **632723:** The EFR32xG22 will limit going to EM1P sleep mode when an 80MHz HRFCO PLL system clock is selected. Going to EM1P sleep is not supported when using the DPLL on this hardware as it can cause clock drift which would impact radio timing and tuning.
- **635237:** Added an event [RAIL_EVENT_PA_PROTECTION](#) to indicate the power protection circuit has kicked in.
- **638067:** Fixed a DMP issue that poached transmit power when switching between protocols using the same channel configuration and channel.
- **638239:** Created a [Callbacks Utility](#) component for application- level callbacks.
- **639833:** Fixed a potential radio hang on a corrupted BLE packet when doing BLE AoX.
- **642893:** Reduced RAIL library flash data alignment needs on the EFR32xG22.
- **645641:** Fixed an EFR32xG22 issue where a state transition to receive after a transmit from EM2 sleep would drop packets.
- **650072:** In multiprotocol RAIL, when the supplied handle is not the active handle, [RAIL_GetRadioState](#) now returns [RAIL_RF_STATE_RX](#) rather than [RAIL_RF_STATE_IDLE](#) if a background receive is currently scheduled.

2.9.2

- **501674:** Reduced the time from last modulated bit on air to ramp down on EFR32xG1x devices.
- **623922:** Fix to set the right power on EFR32xG21 when the PA uses VDD around 1.8V.

2.9.1

- **493409:** Fixed an issue where output power might not be set correctly when changing frequency bands within one channel configuration.
- **497061:** Corrected an issue on EFR32xG1x chips where specifying a large ramp time could erroneously add extra delay to a transmit.
- **519195:** EFR32xG21 will now use RTCC channel 0, as opposed to the PRORTC, to perform sleep timer synchronization. This will help lower the EM2 current consumption for this chip.
- **524046:** Added support for MGM210PB22JIA, MGM210PB32JIA, BGM210PB22JIA and BGM210PA32JIA modules.

2.9.0

- **204557:** Added support for setting the default value of the FramePending bit in outgoing IEEE 802.15.4 data poll ACKs to true. This means that the user would then be responsible for clearing this bit in the frame pending callback instead of having it default to cleared and having to set it. See the [RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks](#) field for configuring this feature.
- **298829:** Added a new API [RAILCb_ConfigSleepTimerSync\(\)](#) to allow for configuration of the PRS and RTCC channels used for timer sync operations.
- **300348:** Added support for a new [RAIL_EVENT_SCHEDULED_RX_STARTED](#) and [RAIL_EVENT_SCHEDULED_TX_STARTED](#), triggered when a scheduled receive or transmit begins. These are the same value because a scheduled receive and transmit cannot occur at the same time. Note: This new event shifted the bit positions of some events in [RAIL_Events_t](#).
- **381485:** Provided new [RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE](#) along with [RAIL_RxChannelHoppingConfigMultiMode_t](#) to configure its parameters. This mode can be configured to tolerate brief loss of timing and/or preamble making it less susceptible to hopping than the single-sense modes. It can also be used with RX duty cycling.
- **402203:** Provided new [RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD](#) to augment each of the [RAIL_RxChannelHoppingMode_t](#) modes with one RSSI Threshold check on entering receive for the channel. If the RSSI is below that specified by [RAIL_RxChannelHoppingConfigEntry_t::rssiThresholdDbm](#) then hop (or if below [RAIL_RxDutyCycleConfig_t::rssiThresholdDbm](#), suspend RX). [RAIL_RxDutyCycleConfig_t](#) has been augmented not only with

this new field, but also now includes [RAIL_RxDutyCycleConfig_t::options](#) field. For those options to be recognized by [RAIL_ConfigRxDutyCycle\(\)](#) (due to backwards compatibility) [RAIL_RxDutyCycleConfig_t::mode](#) must be one of the new WITH_OPTIONS modes, e.g. [RAIL_RX_CHANNEL_HOPPING_MODE_TIMEOUT_WITH_OPTIONS](#).

- **411017:** Relax constraints in RAIL to allow calling [RAIL_SetRxTransitions](#), [RAIL_SetTxTransitions](#), [RAIL_ScheduleRx](#), and all of the [RAIL_BLE_ConfigPhy](#) functions before the radio is completely IDLE.
- **414398:** Added support for the Silicon Labs Power Manager component. When enabled via [RAIL_InitPowerManager\(\)](#) RAIL will communicate directly with the power manager to configure sleep modes. See the [Sleep](#) section for more documentation.
- **429513:** Added support for RAIL's FrameType based length on the EFR32xG22.
- **442248:** Updated the [pa_customer_curve_fits.py](#) helper script to work with Python 3 as well as Python 2.
- **442978:** Changed [pa_customer_curve_fits.py](#) to take maxpower as a parameter to generate better curves. When maxpower and increment are different than the defaults they will now be included in the output curve. The current power curve limits can be read at runtime from the new [RAIL_GetTxPowerCurveLimits](#) API.
- **450750:** The [RAIL_GetRadioEntropy\(\)](#) API will now ensure a valid radio configuration has been loaded using [RAIL_ConfigChannels\(\)](#) since it can cause problems if the radio is used before this.
- **454096:** Updated the function [RAIL_GetRxPacketDetailsAlt](#) to return the time position of the received packet timestamp corresponding to the default location in the packet.
- **454096:** Added a new function, [RAIL_GetTxPacketDetailsAlt2](#), which allows a [RAIL_TxPacketDetails_t](#) structure to be passed as an argument.
- **454096:** Added a new function, [RAIL_GetTxTimePreambleStartAlt](#), which allows a [RAIL_TxPacketDetails_t](#) structure to be passed as an argument.
- **454096:** Added a new function, [RAIL_GetTxTimeSyncWordEndAlt](#), which allows a [RAIL_TxPacketDetails_t](#) structure to be passed as an argument.
- **454096:** Added a new function, [RAIL_GetTxTimeFrameEndAlt](#), which allows a [RAIL_TxPacketDetails_t](#) structure to be passed as an argument.
- **456338:** Fixed an issue with RAIL state transitions where an internal timer wrapping could cause incorrect transition times. This error would previously affect at most one packet every 15 minutes.
- **459581:** Fixed an issue with the EFR32xG21 medium power PA that could cause the output power to be too low for certain power level and ramp time combinations.
- **464534:** Fixed issue where [RAIL_StartAverageRssi\(\)](#) ran twice as long as it should have.
- **464734:** Regenerated the power curves for the EFR32xG22 to allow access to the maximum power level available on the chip.
- **464735:** Closed tiny timing window on EFR32xG13 that might corrupt PTI appended info when idling the radio.
- **465096:** Fixed an issue where [RAIL_Idle\(\)](#) was not properly terminating an ongoing [RAIL_StartAverageRssi\(\)](#) process.
- **465220:** Bluetooth Angle of Arrival and Angle of Departure support has been removed from the EFR32xG13 chip. For future usage of this feature please look to the EFR32xG22.
- **466012:** Fixed an issue where the CRC could be disabled indefinitely on transmit when switching configs in a multiphy setup.
- **474678:** Fixed an issue with duty cycle receive and channel hopping on the EFR32xG1x parts where some parts of the radio would be left on even with long delay parameters causing extra current to be used. This allows for a noticeable improvement in power consumption when using the [RAIL_ConfigRxDutyCycle\(\)](#) API with delays of more than a hundred microseconds.
- **475184:** Fixed an issue on the EFR32xG22 where the receiver was not automatically re-calibrated if the temperature changed significantly while sitting in receive. This could cause the radio to go off channel for significant temperature changes resulting in receive problems.
- **477621:** Improved frequency accuracy on EFR32xG1x devices when the radio configuration has an entry with a large number of channels. Previously, small errors in the channel spacing calculation could exist. If they did, then when computing the channel frequency this error would be multiplied by the channel number minus the start channel for that entry causing some drift for higher order channels. This was not much an issue in most cases, but for certain PHY and crystal combinations it could be worse.
- **477833:** Some radio configurations on the EFR32xG22 are not usable with RAIL address filtering and RAIL 802.15.4 filtering. Add an assert to catch those cases.
- **479539:** Fixed a bug where the [RAIL_ConfigTxPower\(\)](#) would override the PA capacitor tuning values for transmit and receive without caching them. In a multiprotocol scenario this could cause us to apply incorrect PA capacitor tuning values set prior to a call to [RAIL_ConfigTxPower](#). Note that you must still call [RAIL_SetPaCTune](#) after any making any changes to the power configuration via [RAIL_ConfigTxPower](#).
- **479665:** Fixed an issue where [RAIL_SetRxFifo\(\)](#) would not reject a buffer smaller than 64 bytes and would mistakenly think it is very big.

In addition, [RAIL_ASSERT_FAILED_RX_FIFO_BYTES](#) can no longer occur with a 64 byte buffer when doing IR calibration.

- **482007:** Fixed a bug in multiprotocol RAIL where running an IR calibration during a protocol switch would fail. The calibration function will now return [RAIL_STATUS_INVALID_STATE](#) if called in such a scenario.
- **483688:** Fixed an issue where the power mode selected when using [RAIL_TX_POWER_MODE_2P4GIG_HIGHEST](#) on supported chips was not saved in a multiprotocol context and could cause problems.
- **484374:** Fixed regression from 2.8.1 on EFR32xG21 where Bluetooth LE did not include packet sync word on PTI.
- **488752:** Fixed an issue with MGM1x and BGM1x modules that caused PHY changes to be slower than when using stand alone parts. This could cause problems in certain BLE or DMP use cases where reconfiguration time is critical.
- **489214:** Fixed an issue where calling [RAIL_IEEE802154_CalibrateIr2p4Ghz](#), [RAIL_IEEE802154_CalibrateIrSubGhz](#), or [RAIL_BLE_CalibrateIr](#) with a NULL imageRejection parameter would result in a crash.

2.8.10

- **456701:** Fixed an issue on EFR32xG1x parts where calling [RAIL_Init\(\)](#) with the MSC->CTRL.CLKDISFAULTEN bit set would cause a bus fault.
- **702377:** The GB868 PHYs have been modified to improve performance for the EFR32xG1, EFR32xG12, EFR32xG13 and EFR32xG14.

2.8.9

- **645641:** Fixed an issue on EFR32xG22 where a state transition to receive after a transmit from EM2 sleep would drop packets.
- **661651:** Changed RAIL Timer Synchronization over sleep on the EFR32xG21 to use the RTCC instead of the PRORTC to reduce current consumption in EM2.
- **671168:** Fixed an issue on EFR32xG22 parts where current consumption would be higher than expected after radio activity.

2.8.8

- **639833:** Fixed a potential radio hang on a corrupted BLE packet when doing BLE AoX.

2.8.7

- **499216:** Fixed an issue with the EFR32xG21 medium power PA that could cause the output power to be too low for certain power level and ramp time combinations.
- **499366:** Added support for new BGM220 modules.

2.8.6

- **477833:** Some radio configurations on the EFR32xG22 are not usable with RAIL address filtering and RAIL 802.15.4 filtering. Add an assert to catch those cases.
- **484374:** Fixed regression from 2.8.1 on EFR32xG21 where BLE did not include packet sync word on PTI.

2.8.4

- **344182:** Added new [RAIL_BLE_ConfigAoxAntenna](#) API to configure which GPIO pins are used for BLE AoX.
- **411017:** Relax constraints in RAIL to allow calling [RAIL_SetRxTransitions](#), [RAIL_SetTxTransitions](#), [RAIL_ScheduleRx](#), and all of the [RAIL_BLE_ConfigPhy](#) before the radio is completely IDLE
- **465096:** Fixed an issue where [RAIL_Idle\(\)](#) was not properly terminating an ongoing [RAIL_StartAverageRssi\(\)](#) process.
- **467589:** Updated default dynamic multiprotocol (DMP) transition timings to make them work with Zigbee and BLE DMP applications work. The previously suggested workaround of adding 30 us to the default transition time using [RAIL_SetTransitionTime\(\)](#) is no longer required.
- **471373:** Fixed an issue on the EFR32xG22 where loading IEEE 802.15.4 and BLE PHYs without a reset would cause an assert with error code [RAIL_ASSERT_CACHE_CONFIG_FAILED](#).
- **471955:** Fixed an issue with BGM220 modules that caused an assert, [RAIL_ASSERT_INVALID_MODULE_ACTION](#), when using them in previous releases.

2.8.3

- **387749:** Revamped [Features](#) and [rail_features.h](#), providing runtime [RAIL_SupportsSomeFeature\(\)](#) APIs for each of the features as some features may be restricted to certain chips within a family. Also added more consistent [RAIL_SUPPORTS_](#)

compile-time synonyms for the features while retaining the existing `RAIL_FEAT_` defines for backwards compatibility. These defines can now be used in C code and not just preprocessor `#if` statements.

- **442248:** Updated the `pa_customer_curve_fits.py` helper script to work with Python 3 as well as Python 2.
- **464735:** Closed tiny timing window on EFR32xG13 that might corrupt PTI appended info when idling the radio.
- **464774:** * Calling `RAIL_ConfigSleep()` with `RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED` on chips that use the PRORTC for synchronization (EFR32xG13 and newer) will now only configure the choose the LF clock source if the PRORTC IRQ is disabled. This allows for other code to safely configure the PRORTC like the Silicon Labs generic sleep timer.
- **466396:** Added support for new BGM220 modules.
- **469015:** Fixed an issue on the EFR32xG21 that could cause the `RAIL_GetRadioEntropy()` function to return the same first 4 bytes when called with the radio off after a reset.

2.8.2

- **456338:** Fixed an issue with RAIL state transitions where an internal timer wrapping could cause incorrect transition times. This error would previously affect a maximum of one packet every 15 minutes.
- **460062:** Fixed a `RAIL_ScheduleRx()` issue where `RAIL_EVENT_RX_SCHEDULED_RX_END` might not be posted when the Rx `RAIL_StateTransitions_t::error` transition is to `RAIL_RF_STATE_IDLE` and the Rx window ended during receipt of an erroneous packet.

2.8.1

- **251720:** Added the new `RAIL_STREAM_10_STREAMRAIL_StreamMode_t` to allow you to send a 1010 stream for debugging.
- **316106:** Added a new function, `RAIL_StartTxStreamAlt`, which allows the specific antenna to be specified for a stream transmit.
- **383197:** Added new `RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE` packet handle to allow the user to get a reference to the oldest unreleased complete packet.
- **435689:** Added a new `External Thermistor` interface to RAIL. This allows access the user to connect and read the impedance of an external thermistor on supported chips.
- **444205:** Fixed a transmit-from-idle issue with `RAIL_StartCcaCsmaTx()` or `RAIL_StartCcaLbtTx()` which would always fail when the `RAIL_StateTiming_t::idleToRx` is configured below the minimum the radio is capable of achieving (typically 65-100 microseconds depending on platform).
- **450750:** The `RAIL_GetRadioEntropy()` API will now ensure a valid radio configuration has been loaded using `RAIL_ConfigChannels()` since it can cause problems if the radio is used before this.
- **452628:** Fixed an issue where idling the radio from an Rx antenna diversity mode would consume extra power.
- **452690:** Fixed an issue where Rx antenna diversity could be left enabled after switching to a radio configuration that lacks diversity support.
- **455495:** Added `RAIL_IEEE802154_ConvertRssiToEd()` and `RAIL_IEEE802154_ConvertRssiToLqi()` to assist Zigbee 802.15.4 certification testing.
- **456225:** Changed the value of `RAIL_FREQUENCY_OFFSET_INVALID` from -1 to -32768 since -1 is a reasonable frequency offset to pass to `RAIL_SetFreqOffset()`. Also added convenience definitions `RAIL_FREQUENCY_OFFSET_MIN` and `RAIL_FREQUENCY_OFFSET_MAX` to specify the valid range of offset values the radio supports.

2.8.0

- **197573:** Suppressed extraneous `RAIL_EVENT_TX_START_CCA` events that might occur during long CCA durations. Now only one such event should occur per CCA try.
- **295265:** Added two new APIs, `RAIL_GetSyncWords` and `RAIL_ConfigSyncWords()`, to allow getting and setting the sync word configuration of your PHY at runtime.
- **301472:** On EFR32xG12 thru EFR32xG14, added support for 802.15.4G-2012 SUN PHY dynamic frame payload whitening on reception and transmit based on the PHY header's Data Whitening flag setting. This feature is automatically enabled when `RAIL_IEEE802154_ConfigGOptions()` `RAIL_IEEE802154_G_OPTION_GB868` is enabled, and assumes the radio configuration specifies the appropriate whitening algorithm and settings.
- **301488:** On EFR32xG12 thru EFR32xG14, added support for 802.15.4G-2012 SUN PHY dynamic frame payload 2/4-byte FCS (CRC) on reception and transmit based on the PHY header's FCS Type flag setting. This feature is automatically enabled when `RAIL_IEEE802154_ConfigGOptions()` `RAIL_IEEE802154_G_OPTION_GB868` is enabled. The radio configuration's (single) CRC algorithm settings are ignored, overridden by RAIL.
- **301501:** On EFR32xG12 thru EFR32xG14, 802.15.4 AutoACK behavior has also been updated so transmitted immediate ACKs reflect the Whitening and 2/4-byte FCS of the received frame being acknowledged.

- **369722:** Added [RAIL_TX_OPTION_CCA_ONLY](#) to just perform CCA (CSMA/LBT), stopping short of automatically transmitting when the channel is clear.
- **369727:** Added support for a new [RAIL_EVENT_TX_STARTED](#), triggered when the first preamble bit is about to go on-air. Also included the ability to retrieve the equivalent [RAIL_PACKET_TIME_AT_PREAMBLE_START](#) timestamp of that event from the event's handler via [RAIL_GetTxTimePreambleStart\(\)](#). Note: This new event shifted the bit positions of some events in [RAIL_Events_t](#).
- **369727:** Also changed [RAIL_GetRxTimePreambleStartAlt\(\)](#), [RAIL_GetRxTimeSyncWordEndAlt\(\)](#), and [RAIL_GetRxTimeFrameEndAlt\(\)](#) to properly update its pPacketDetails' [RAIL_PacketTimeStamp_t::timePosition](#) to reflect the adjusted [RAIL_PacketTimeStamp_t::packetTime](#) rather than leaving it as [RAIL_PACKET_TIME_DEFAULT](#).
- **384878:** Clarified documentation of the [RAIL_EVENT_RX_ACK_TIMEOUT](#) event and [RAIL_AutoAckConfig_t::ackTimeout](#) period which extends only to packet sync word detection of an expected ACK, not packet completion of that ACK.
- **397072:** Added an API, [RAIL_StopInfinitePreambleTx](#), that can stop an infinite preamble on PHYs configured to use infinite preambles.
- **402991:** Added additional information to the packet trace stream for the Z-Wave protocol to indicate what region is currently active to help with decoding.
- **406080:** Added support for RFSense Selective (OOK) mode for supported chips, which currently includes only EFR32xG22 devices. Please refer to RAIL internal chip specific documentation for more details.
- **406871:** Documented RAIL's internal 16-packet metadata FIFO which exists on EFR32 platforms supplementing the receive FIFO of packet data. Refer to [Data Management](#) and [EFR32](#) for details. Included is support for a new [RAIL_EVENT_RX_FIFO_FULL](#), triggered with any packet completion event in which the receive FIFO or packet metadata FIFO are full. This tells the application it must free up the oldest packets/data ASAP to reduce the chance of [RAIL_EVENT_RX_FIFO_OVERFLOW](#) (however, overflow may already have occurred). Note: This new event shifted the bit positions of some events in [RAIL_Events_t](#).
- **411498:** [RAIL_StartAverageRssi\(\)](#) now returns [RAIL_STATUS_INVALID_STATE](#) if called when the radio is not idle, enforcing its documented behavior.
- **414114:** Added a new PA mode which will attempt to automatically choose the PA which consumes the least amount of current to reliably produce the requested output power. See [RAIL_EnablePaAutoMode\(\)](#) for details.
- **417340:** Fixed an issue where [RAIL_RxPacketDetails_t::isAck](#) would incorrectly be set true for non-ACK or unexpected ACK packets received successfully (e.g. when [RAIL_IEEE802154_ACCEPT_ACK_FRAMES](#) is enabled) or aborted while waiting for the expected ACK. Note that when [RAIL_RX_OPTION_IGNORE_CRC_ERRORS](#) is in effect, an expected ACK includes one that fails CRC, and will have [isAck](#) set true.
- **418493:** [RAIL_ConfigRadio](#) will now return [RAIL_STATUS_INVALID_STATE](#) if called from the inactive config in dynamic multiprotocol instead of returning success but not applying the change.
- **427934:** Fixed a race condition that could cause a device to not re- enable frame detection after an Rx overflow event if the overflow was processed and cleared very quickly.
- **430026:** Enforced and clarified that [RAIL_Init\(\)](#) should not be called more than once per protocol.
- **430081:** Fixed an issue where the first Clear Channel Assessment (CCA) of a CSMA/LBT transmit from radio idle state would consistently fail when the [RAIL_CsmaConfig_t::ccaBackoff](#) or [RAIL_LbtConfig_t::lbtBackoff](#) time is smaller than the [RAIL_StateTiming_t::idleToRx](#) time.
- **436163:** Fixed a post-receive transition timing issue for received packets that were on the air longer than 32 milliseconds. AutoACK turnaround timing should now behave properly at low data rates.
- **437054:** Fixed an issue with the [pa_customer_curve_fits.py](#) that caused values below -12 to not be considered when computing the fit. Re-generated default, Silicon Labs-provided curves to consume this fix, resulting in minor changes to the lowest-power segment in curve-fit based PA's. If using a custom power curve created using the documentation in AN1127 customers should re-run the script on the already collected output data to get slightly more accurate curves.
- **441635:** Return the correct [RAIL_TxPowerMode_t](#) value of [RAIL_TX_POWER_MODE_NONE](#) from [RAIL_GetTxPowerConfig](#) if called before [RAIL_ConfigTxPower](#).
- **446289:** Fixed [RAIL_IDLE_ABORT](#) to idle the radio sooner when in [RAIL_RF_STATE_RX](#), especially now that [RAIL_RxChannelHoppingConfigEntry_t::delay](#) can extend the time in that state.
- **447578:** Fixed an issue where setting a transmit power over the maximum allowed for a given channel would result in no change in the output power instead of using the maximum allowed value.
- **450187:** Fixed an issue where calling [RAIL_Idle\(\)](#) with [RAIL_IDLE_FORCE_SHUTDOWN](#) while in receive with channel hopping enabled could corrupt some internal channel hopping state and trigger a bus fault or other radio problems.

2.7.4

- **436215:** Fixed an issue on the EFR32xg13 parts that caused the 2Mbps BLE IFS time to be incorrect when receiving very short packets.
- **443144:** Fixed the module specific config libraries for the xGM210xA modules: BGM210LA22JIF, BGM210LA22JNF, BGM210PA22JIA, BGM210PA22JNA, BGM210PA32JIA, BGM210PA32JNA, MGM210LA22JIF, MGM210LA22JNF, MGM210PA22JIA, MGM210PA22JNA, MGM210PA32JIA, MGM210PA32JNA. Without this change these modules would assert with [RAIL_ASSERT_INVALID_MODULE_ACTION](#).

2.7.3

- **425464:** Improved interoperability with many common devices for the 1Mbps Viterbi BLE PHY on the EFR32xG13 chips. Note that this does revert a previous change to improve interoperability from [changeloglist_2_6_4](#) that attempted to improve interoperability with a few specific devices. For best interoperability we recommend using the [RAIL_BLE_ConfigPhy1Mbps\(\)](#) standard PHY.
- **431229:** Resolved an issue on EFR32xG1x devices which could cause the transmit power to be incorrect when using the low power 2.4GHz PA ([RAIL_TX_POWER_MODE_2P4_LP](#)).

2.7.2

- **336739:** Added new [RAIL_EVENT_RX_SCHEDULED_RX_MISSED](#) and [RAIL_EVENT_TX_SCHEDULED_TX_MISSED](#) events to inform the user when these are missed after waking from [RAIL_Sleep\(\)](#). This can happen if the specified wakeup time is not long enough.
- **421410:** Fixed an issue introduced in 2.7.1 with TX PTI on EFR32xG13 including an extra byte of appended information, confusing the network analyzer.

2.7.1

- **337468:** Fixed an issue where calling [RAIL_Sleep\(\)](#) if there were no RAIL events pended would not stop and synchronize the clock source as requested but would return success indicating it had. The clock source will now be properly stopped and synchronized even if there are no events pending and it will be the user's responsibility to wake up on time.
- **337468:** Fixed an issue where calling [RAIL_Wake\(\)](#) without a successful call to [RAIL_Sleep\(\)](#) first could cause the clock source to drift even when using [RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED](#).
- **389462:** Fixed an issue with [RAIL_Calibrate\(\)](#) in multiprotocol, which would return [RAIL_STATUS_INVALID_STATE](#) if it is called with an inactive railHandle. Now, [RAIL_Calibrate\(\)](#) will make the given railHandle active, if not already, and perform calibration.
- **389462:** Fixed an issue with [RAIL_Calibrate\(\)](#) where after completing calibration the radio would no longer be able to receive any packets, sometimes, until it was reset.
- **396786:** Added support for the following EFR32xG21-based modules: BGM210L022JIF1, BGM210L022JIF2, BGM210L022JNF1, BGM210L022JNF2, BGM210LA22JIF1, BGM210LA22JIF2, BGM210LA22JNF1, BGM210LA22JNF2, BGM210P022JIA1, BGM210P022JIA2, BGM210P022JNA2, BGM210P032JIA1, BGM210P032JIA2, BGM210P032JNA2, BGM210PA22JIA1, BGM210PA22JIA2, BGM210PA22JNA2, BGM210PA32JIA1, BGM210PA32JIA2, BGM210PA32JNA2, MGM210L022JIF1, MGM210L022JIF2, MGM210L022JNF1, MGM210L022JNF2, MGM210LA22JIF1, MGM210LA22JIF2, MGM210LA22JNF1, MGM210LA22JNF2, MGM210P022JIA1, MGM210P022JIA2, MGM210P022JNA2, MGM210P032JIA1, MGM210P032JIA2, MGM210P032JNA2, MGM210PA22JIA1, MGM210PA22JIA2, MGM210PA22JNA2, MGM210PA32JIA1, MGM210PA32JIA2, MGM210PA32JNA2
- **401826:** Fixed several issues and race conditions with [RAIL_ScheduleRx\(\)](#) window end handling and event reporting to conform to its intended design:
 - When the RX window ends with no [RAIL_EVENTS_RX_COMPLETION](#) imminent, [RAIL_EVENT_RX_SCHEDULED_RX_END](#) is posted.
 - When the RX window is implicitly ended by one of the [RAIL_EVENTS_RX_COMPLETION](#), e.g. because it results in an RX transition to Idle or because [RAIL_ScheduleRxConfig_t::rxTransitionEndSchedule](#) is non-zero, the event(s) posted depend on that setting:
 - When zero, both events are posted simultaneously.
 - When non-zero, only the appropriate [RAIL_EVENTS_RX_COMPLETION](#) is posted, unless that event is not enabled, in which case [RAIL_EVENT_RX_SCHEDULED_RX_END](#) is substituted instead.
 - When the RX window ends while receiving a packet, it is deferred to the expected RX completion event (which includes aborting that packet when [RAIL_ScheduleRxConfig_t::hardWindowEnd](#) is non-zero). Event(s) reported at that time are the same as in the previous case.

- **406276:** Restored ability for [RAIL_StartCcaCsmaTx\(\)](#) and [RAIL_StartCcaLbtTx\(\)](#) to perform an immediate transmit when their respective [RAIL_CsmaConfig_t::csmaTries](#) or [RAIL_LbtConfig_t::lbtTries](#) is 0. This functionality was improperly removed in 2.6.0.
- **407047:** Fixed an EFR32xG21 issue where CSMA/LBT CCA durations were significantly shorter than specified.
- **408096:** Fixed the 802.15.4 ACK turnaround time on the EFR32xG21 platform. Due to a calculation error this was actually 18 us too short which could cause interoperability problems.
- **414257:** Added [RAIL_TX_POWER_LEVEL_MAX](#) which can be used to set the max PA power level across PA's. [RAIL_TX_POWER_LEVEL_INVALID](#) was added in 2.7.0 as the value 255. Some customers were using 255 to set max power across PA's with [RAIL_SetTxPower](#), which previously worked, but will now return an error.

2.7.0

- **379150:** Added support to [RAIL_Sleep\(\)](#) for the PLFRCO on EFR32xG13 Rev D parts.
- **357922:** Changed the LQI metric for the 2.4GHz IEEE802.15.4 PHY configurations to be scaled from 0 - 255 and to include more data to make it more stable. This can impact existing applications that are using the LQI values returned in prior RAIL versions.
- **378035:** Use of the unsafe enum `GPIO_Port_TypeDef` within RAIL aggregate types [RAIL_PtiConfig_t](#) and [RAIL_AntennaConfig_t](#) has been replaced by safe `uint8_t`.
- **376229:** Fixed an issue with Rx antenna diversity operation that prevented CCA from working, causing CSMA failures.
- **315849:** Reduced switch time overhead for dynamic multiprotocol applications. The new switch times as well as information about them are documented at [Understanding the Protocol Switch Time](#).
- **370805:** Fixed an issue with the EFR32xG21 reporting a phantom packet on PTI after reset.
- **377026:** Improved documentation of [RAIL_RxPacketStatus_t](#) values and their corresponding [RAIL_Events_t](#) events.
- **400303:** Corrected an issue where [RAIL_EVENT_TX_CHANNEL_BUSY](#) due to [RAIL_CsmaConfig_t::csmaTimeout](#) or [RAIL_LbtConfig_t::lbtTimeout](#) would prevent further transmits.
- **400303:** Corrected an issue where an invalid [RAIL_CsmaConfig_t::ccaDuration](#) or [RAIL_LbtConfig_t::lbtDuration](#) too large for the radio configuration to handle would not fail the respective transmit; this now returns [RAIL_STATUS_INVALID_PARAMETER](#).
- **400303:** Clarified that [RAIL_EVENT_TX_BLOCKED](#) and [RAIL_EVENT_TX_CHANNEL_BUSY](#) leave the TX FIFO intact without consuming any of its packet data.
- **392350:** Corrected an issue where the radio might be left in receive after a [RAIL_EVENT_TX_BLOCKED](#) or [RAIL_EVENT_TX_CHANNEL_BUSY](#) when the transmit [RAIL_StateTransitions_t::error](#) is [RAIL_RF_STATE_IDLE](#).
- **251786:** Added new 802.15.4 RAIL APIs [RAIL_IEEE802154_EnableEarlyFramePending\(\)](#) and [RAIL_IEEE802154_EnableDataFramePending\(\)](#) to support Thread Basil-Hayden Enhanced Frame Pending feature.
- **369736:** Added new 802.15.4 RAIL APIs [RAIL_IEEE802154_ConfigGOptions\(\)](#) and [RAIL_IEEE802154_ConfigEOptions\(\)](#) for configuring certain 802.15.4E-2012 and G-2012 features needed by GB868. Also, RAIL may now suppress the [RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND](#) when latency has delayed the handling of that event too late to influence the outgoing ACK for it.
- **385960:** Added information to packet trace for every protocol switch in dynamic multiprotocol telling the user what protocol RAIL has changed to as well the radio event that triggered this switch. This information is visible in Network Analyzer for better debugging of DMP applications.
- **389439:** The EFR32xG13 AoX RAIL library is no longer present as that functionality has been merged into the standard RAIL library for EFR32xG13.
- **360371:** Fixed an issue where calling [RAIL_GetTxPowerDbm](#) prior to calling [RAIL_SetTxPower](#) would return -500 i.e. -50dBm. As a part of the fix, we now return an invalid dBm value, [RAIL_TX_POWER_MIN](#), if [RAIL_SetTxPower](#) was not called before calling [RAIL_GetTxPowerDbm](#) or if [RAIL_SetTxPower](#) it returns an error status.
- **382575:** Added a new API [RAIL_UseDma\(\)](#), which can be used to enhance RAIL startup speed if called before [RAIL_Init\(\)](#).
- **376711:** Added support for Z-Wave node ID based packet filtering via the [RAIL_ZWAVE_OPTION_NODE_ID_FILTERING](#) option.
- **386163:** Added support for the radio sending an ACK packet automatically when in Z-Wave mode as long as node ID filtering and [Auto-Ack](#) features are enabled and a packet requesting an ACK is sent to the device.

2.6.5

- Corrected the frequencies used by Z-Wave in the China, Malaysia, and India regions to match the specification. China was using the EU frequencies, Malaysia was using 868.10 MHz, and India was not applying the required 20 kHz offset to the R1 channel.

2.6.4

- Fix an issue on the EFR32xG21 where the medium power and low power PAs might not complete ramping and end up transmitting at a lower power than desired.
- Improved interoperability of the BLE 1Mbps PHY on EFR32xG13 devices.
- Improved performance of OOK PHYs.
- Updated packet trace format to include extra information for BLE packets.

2.6.3

- Fixed an issue introduced in 2.6.2 where issuing a [RAIL_Idle\(\)](#) with mode [RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS](#) could sometimes cause the receive FIFO to be corrupted and trigger a RAIL assert with code ([RAIL_ASSERT_FAILED_ILLEGAL_RXLEN_ENTRY_STATUS](#)).
- Various bug fixes and improvements for the EFR32xG21 parts.

2.6.2

- Updated the PA curves on the EFR32xG21 chips to slightly improve transmit power accuracy.
- Resolved an issue where [RAIL_GetChannelHoppingRssi](#) would return an RSSI value even if receive was disabled.
- Resolved an issue where [RAIL_GetChannelHoppingRssi](#) would not update the RSSI value while receiving packets.
- Resolved an issue in multiprotocol where an absolute event with a non zero slip time would always work even if scheduled in the past.

2.6.1

- Resolved an issue on the EFR32xG21 platform that could cause the chip to hang when coming out of a reset if [RAIL_Idle\(\)](#) was called at the wrong time.
- Added [RAIL_GetChannelHoppingRssi](#) API to allow the user to read the RSSI from each channel when channel hopping is enabled.
- Fixed an issue where CRC-failed packets received with [RAIL_RX_OPTION_IGNORE_CRC_ERRORS](#) would be reported with [RAIL_RX_PACKET_READY_SUCCESS](#) instead of [RAIL_RX_PACKET_READY_CRC_ERROR](#) when the [RAIL_EVENT_RX_FRAME_ERROR](#) event isn't enabled.
- Fixed an issue on the EFR32xG21 platform that caused the chip to incorrectly compute the PA ramp time at high power levels. If this happened, the ramp time would be higher than expected and cause large delays in the [RAIL_StateTiming_t::rxToTx](#) time which is used for sending ACK packets.
- Fixed an issue on the EFR32xG21 platform that could cause erroneous Rx overflow events and failed receives when using the 125Kbps BLE coded PHY.

2.6.0

- Fixed an issue causing the receive antenna to always be reported as antenna 0 when using receive antenna diversity.
- Added alternative APIs for calculating receive timestamps, which give correct results for Long-Range BLE: [RAIL_GetRxTimeFrameEndAlt](#), [RAIL_GetRxTimePreambleStartAlt](#), [RAIL_GetRxTimeSyncWordEndAlt](#).
- Fixed an issue where receive events could corrupt the transmit packet length in frame type based length mode if they came in between the start transmit call and the actual transmit.
- Fixed an Auto-ACK problem where ACKs were being sent for CRC error frames when [RAIL_RX_OPTION_IGNORE_CRC_ERRORS](#) is in effect.
- Fixed a math error that could cause the [RAIL_CsmaConfig_t::ccaDuration](#) to be shorter than requested. This would happen whenever the (ccaDuration * rx_baudrate) exceeded the size of a 32 bit integer.
- Fixed an issue in the multiprotocol RAIL library where the [RAIL_GetTxPower\(\)](#) function could return the wrong output power if called with any RAIL handle other than the active one. We will now return the actual power that would be set if that protocol is active.
- Fixed the PA conversion code so that it better handles error conditions and will not trigger a fault when given NULL parameters.
- Added new API to retrieve the channel index of the most recent Beam packet accepted: [RAIL_ZWAVE_GetBeamChannelIndex](#).
- Added a new [RAIL_DelayUs\(\)](#) API for short blocking delays using the RAIL time base.

- Fixed a problem where state timings would not properly be updated when switching if both protocols used the same channel configuration structure.
- Fixed an issue on the EFR32xG14 series that prevented Rx antenna diversity from functioning properly. This was introduced in the RAIL 2.5.0 release.
- Restored pre-2.5.0 behavior with [RAIL_SetRxFifoThreshold\(\)](#) to disable threshold events from occurring when passed an rxThreshold of the FIFO's size or larger. A 2.5.0 change allowed such rxThreshold settings to cause the event to fire when the buffer became full, though it's been mis-documented as one byte away from becoming full for quite a while – something that was never actually implemented and is now removed from the documentation.
- Fixed [RAIL_StartCcaCmaTx\(\)](#) and [RAIL_StartCcaLbtTx\(\)](#) to return [RAIL_STATUS_INVALID_PARAMETER](#) when passed invalid configuration parameters in their respective [RAIL_CsmaConfig_t](#) or [RAIL_LbtConfig_t](#), whose documentation has been updated accordingly. In earlier releases the requested operation would proceed with an invalid or incomplete hardware configuration, resulting in undefined behavior.

2.5.1

- Worked around a BLE longrange problem where Rx abort could cause deafness.

2.5.0

- Added a channel hopping feature with [RAIL_ConfigRxChannelHopping](#) and [RAIL_EnableRxChannelHopping](#). This feature allows users to specify a list of channels to scan over during receive. This feature allows for much faster scanning with less delay (about 100us) than simply repeatedly calling [RAIL_StartRx\(\)](#) with different channels.
- Included two default transmit power curves in the library: [RAIL_TxPowerCurvesDcdc](#) and [RAIL_TxPowerCurvesVbat](#). When the pa-conversions plugin is used, one of these can be passed to the plugin, based on the way the power amplifier is powered.
- Added the ability to specify a Link Quality Indicator (LQI) conversion callback with [RAIL_ConvertLqi](#) such that the hardware's LQI value can be changed before application consumption of that value.
- Fixed an issue where [RAIL_StopTxStream\(\)](#) was taking an extended amount of time to complete.
- Certain received packet details in [RAIL_RxPacketDetails_t](#), which were formerly available only for successfully received packets, are now also available for unsuccessfully received packets.
- [RAIL_DataMethod_t](#) and related documentation have been updated to reflect that many FIFO_MODE concepts can now be used in PACKET_MODE, and the distinction between these modes has become nearly invisible for transmit. Also eliminated the previous restriction that both receive and transmit [RAIL_DataMethod_t](#) must match in the [RAIL_DataConfig_t](#) passed to [RAIL_ConfigData\(\)](#).
- [RAIL_ConfigData\(\)](#) no longer resets the receive FIFO when switching to [RAIL_DataMethod_t::FIFO_MODE](#).
- Added a new [RAIL_TxOptions_t](#) option [RAIL_TX_OPTION_CCA_PEAK_RSSI](#) to allow the user to use the peak RSSI instead of the average RSSI over the CCA period during CSMA and LBT transmits. This is only supported on EFR32xG12 and newer silicon.
- Added a new [RAIL_TxOptions_t](#) option [RAIL_TX_OPTION_ALT_PREAMBLE_LEN](#) to allow the user to transmit with a runtime-specified preamble length. This alternate preamble length is set with [RAIL_SetTxAltPreambleLength](#).
- Added [Z-Wave](#) PHY configurations and functionality to the RAIL library.
- Fixed an issue that could cause register corruption in very rare conditions due to a bus access race condition. The specific registers impacted were limited to those related to the radio and the corruption was most likely when running the coherent IEEE802.15.4 PHY on the EFR32xG12 and EFR32xG13 platforms. On other platforms and PHY configurations the issue is possible though very unlikely and never observed (id: 317234).
- Resolved a multiprotocol scheduler issue that could cause some interrupted events to not be reported to the caller. Since these events never run there are no other related events either so state machines relying on RAIL events could get stuck in invalid states (id: 341124).
- Enhanced [RAIL_SetRxFifoThreshold\(\)](#) and [RAIL_SetTxFifoThreshold\(\)](#) to now support a threshold value of [RAIL_FIFO_THRESHOLD_DISABLED](#) to explicitly disable a threshold previously established.
- Fixed a documentation issue where a variety of RAIL APIs were incorrectly shown as "Referenced by [RAIL_CopyRxPacket\(\)](#)".

2.4.1

- Fixed an issue with [RAIL_Sleep\(\)](#) that could cause the wake event to be set very far in the future. This would prevent waking at the proper time and cause problems for the application.
- Fixed an issue in the multiprotocol library where the txToRxSearch time set by [RAIL_SetStateTiming\(\)](#) would be lost after a protocol switch if there were no timings stored in the channel configuration.
- Fixed an issue preventing radio coexistence transmit abort counter from incrementing.
- Fixed radio coexistence HAL_CONFIG support for shared priority and shared request.

2.4.0

- Added ability to set custom receive FIFO sizes via the [RAIL_SetRxFifo\(\)](#) function. The new [RAILCb_SetupRxFifo\(\)](#) callback may be used to override the default FIFO initialization to save RAM.
- Added alternate functions to get packet details, in a way that helps lower code size: [RAIL_GetRxPacketDetailsAlt\(\)](#) and [RAIL_GetTxPacketDetailsAlt\(\)](#). These come with functions to adjust the returned time stamp to different places in the packet: [RAIL_GetRxTimePreambleStart\(\)](#), [RAIL_GetRxTimeSyncWordEnd\(\)](#), [RAIL_GetRxTimeFrameEnd\(\)](#), [RAIL_GetTxTimePreambleStart\(\)](#), [RAIL_GetTxTimeSyncWordEnd\(\)](#), and [RAIL_GetTxTimeFrameEnd\(\)](#).
- Broke up the IR calibration code to save code size for Zigbee and BLE where the results are stored on chip.
- Added a new [RAIL_SetRssiOffset\(\)](#) command to offset all RSSIs used by RAIL to match any hardware offsets on a given board.
- Added a new [RAIL_StopTx\(\)](#) function to stop only pending or current transmits without impacting receive.
- For multiprotocol, added the ability to update a task's priority after it started with [RAIL_SetTaskPriority\(\)](#).
- Added the ability to pre-apply a radio configuration to hardware before a TX or RX operation begins via [RAIL_PrepareChannel\(\)](#).
- Added the ability to query RAIL for the channel configuration currently configured for use via [RAIL_GetChannel\(\)](#).
- Fixed an issue where relative tasks with a non-zero slip time could start a few hundred microseconds earlier than they were supposed to.
- Fixed an issue that could cause antenna diversity to not function correctly.
- Fixed an issue where [RAIL_Idle\(\)](#) with [RAIL_IDLE_FORCE_SHUTDOWN](#) could cause an erroneous [RAIL_EVENT_TX_ABORTED](#) event to fire.

2.3.1

- Fixed one case where calling [RAIL_Idle\(\)](#) with the mode set to [RAIL_IDLE_FORCE_SHUTDOWN](#) could cause us to output partial packet trace appended information.
- Deferred multiprotocol event failures until just before the event must start when the expected runtime of a task causes it to not fit. These were previously reported immediately even though future tasks could change.

2.3.0

- Added a [RAIL_CopyRxPacket\(\)](#) helper for use after [RAIL_GetRxPacketInfo\(\)](#).
- Added a fast, but inconsistent, transition times mode. This can be activated by setting state timings to 0.
- Resolved a potential receive FIFO corruption that could occur when calling [RAIL_HoldRxPacket\(\)](#) or [RAIL_PeekRxPacket\(\)](#).
- A multi-timer API was added that allows for multiple software timers all driven off the one hardware timer. This must be explicitly enabled and will use more code than the standard APIs, but can be done through [RAIL_ConfigMultiTimer\(\)](#). Once in use, all timer APIs must use the multi timer versions to prevent conflicts with the hardware timer.
- Added diagnostic functions [RAIL_Verify\(\)](#) and [RAIL_ConfigVerification\(\)](#) to provide a means of verifying internal radio memory contents.
- Added alternative calibration APIs to [RAIL_Calibrate](#), for potential code size savings: [RAIL_CalibrateTemp](#), [RAIL_CalibrateIr](#), and [RAIL_ApplyIrCalibration](#).
- Fixed an issue where requesting a small fixed CSMA backoff could result in a very large value from being selected instead.
- Made PHY configuration improvements for BLE and Zigbee on the efr32xg14 platform.

2.2.0

- In FIFO mode on the EFR32 the FIFO could only be read after 8 bytes were received. This restriction has now been lifted though the user must be careful and understand that in certain cases they could be reading appended information and not packet bytes.
 - On the EFR32XG14 some BLE packets showed up with invalid timestamps. These are now filtered and reported as aborted to the user so that the stack does not time sync to an invalid timestamp and lose the connection.
 - Fixed an issue where state transition times would not be recomputed when changing the PA ramp time on the fly.
 - Fixed a regression in 2.1.1 that caused Frame Type based length configurations to lock up the receiver. There was an issue where bits in the byte that were not supposed to be a part of the frame type were being included some times and could cause problems in the receive processing logic.
 - Prevented [RAIL_StartRx\(\)](#) from resetting the receive FIFO. This was a legacy feature that was not documented and, though it generally worked, would cause problems to users processing deferred receive events or users that called [RAIL_StartRx\(\)](#) in an interrupt disabled context. If this behavior was required it can be replicated by idling the radio and then calling [RAIL_ResetFifo\(\)](#) with `rxFifo` set to true before any calls to [RAIL_StartRx\(\)](#).
 - Fixed an issue in the [RAIL_IIEEE802154_SetAddresses\(\)](#) function that caused it to never set any long address except for the first one and to set that one to the value given in the final long address entry.
 - Updated the documentation for the IEEE 802.15.4 and BLE protocol functions to show proper usage with the new RAIL 2.0 API.
 - Dynamic Multiprotocol fixes:
 - Added support for each protocol having auto ACK enabled by allocating space for independent ACK buffers.
 - Ensured that the sync word is properly output in BLE Long Range no matter when the packet data is loaded into the transmit FIFO.
 - Prevented the sync word in packet trace feature from being turned on in Zigbee when it's enabled by BLE.
-

2.1.2

- Removed unimplemented functions [RAIL_GetActiveChannelConfig\(\)](#) and [RAIL_GetActiveChannelConfigEntry\(\)](#).

2.1.1

- [RAIL_PauseTxAutoAck\(\)](#) was internally pausing the Rx AutoAck logic. This has now been corrected.
- EFR32XG12 and EFR32XG13 devices were detecting a high number of false 15.4 sync words. Even though these packets were eventually filtered they would show up on PTI. In 2.1.1 there should be significantly fewer of these packets as some of this was caused by a bug in a configuration algorithm.
- Channel power limits for BLE PHYs were not always being respected. This could have caused us to use a power which might result in certification issues.
- The [RAIL_StateBuffer_t](#) type was incorrectly sized resulting in a waste of RAM for RAIL applications. This has now been trimmed down to the actual size required.

2.1.0

- Packet receive APIs were totally revamped to allow faster and out-of-order deferred processing of received packets and their data outside of interrupt/callback context. Gone are [RAIL_GetRxPacketHandle\(\)](#), [RAIL_ReadRxFifoAppendedInfo\(\)](#), and the memory management [RAILCb_](#) callbacks, replaced with [RAIL_GetRxPacketInfo\(\)](#) that provides pointers directly to the received data held in RAIL's internal receive FIFO buffer allowing for zero-copy operation. See other RAIL 2.x porting documentation and new routines [RAIL_GetRxPacketInfo\(\)](#), [RAIL_GetRxPacketDetails\(\)](#), [RAIL_HoldRxPacket\(\)](#), and [RAIL_ReleaseRxPacket\(\)](#), along with updated [RAIL_PeekRxPacket\(\)](#) and [RAIL_ReadRxFifo\(\)](#) behavior.
- On the transmit side, zero-copy is also now supported by allowing applications to specify their own buffers for the transmit FIFO, eliminating RAIL's former internal fixed-size 512-byte one. See new [RAIL_SetTxFifo\(\)](#).
- The former [RAIL_LoadTxData\(\)](#) has been merged into an updated [RAIL_WriteTxFifo\(\)](#) API.
- The [RAIL_Event_t](#) type was renamed [RAIL_Events_t](#) now that multiple events can be presented to the application simultaneously in RAIL's generic events callback, rather than relying on the internal arbitrary order that RAIL imposed previously, which may not match the application's or protocol's expectations.
- Event/callback configuration is now managed entirely by [RAIL_ConfigEvents\(\)](#), eliminating the old [RAIL_ConfigTx\(\)](#) and [RAIL_SetCallbackConfig\(\)](#) APIs.
- Channel numbers were increased in range from 8 bits to 16 bits, and channel configurations made much more flexible allowing for multi-band radio configurations and an easy way to specify TX power limits on certain channels that otherwise

share a common configuration. See new APIs [RAIL_GetActiveChannelConfig\(\)](#) and [RAIL_GetActiveChannelConfigEntry\(\)](#) along with changes to [RAIL_ConfigChannels\(\)](#).

- Power levels were given a formal [RAIL_TxPowerLevel_t](#) type, and [RAIL_SetTxPower\(\)](#) now returns a [RAIL_Status_t](#) not the power level actually achieved – must now use [RAIL_GetTxPower\(\)](#) to get that.
- PA power curves and the conversion between RAIL power level and dBm have been fully abstracted to live outside of RAIL, allowing customers complete flexibility to characterize their designs and PA selection. See new APIs [RAIL_ConfigTxPower\(\)](#), [RAIL_ConvertDbmToRaw\(\)](#), [RAIL_ConvertRawToDbm\(\)](#), [RAIL_GetTxPowerConfig\(\)](#), [RAIL_GetTxPowerDbm\(\)](#), [RAIL_SetTxPowerDbm\(\)](#), and [RAIL_EnablePaCal\(\)](#).
- Protocols using Frame Types to determine packet length are better integrated into the Studio radio configurator and channel configuration. Gone are APIs like [RAIL_ConfigFrameTypeLength\(\)](#) and [RAIL_EnableAddressFilterByFrameType\(\)](#) whose functionality is now subsumed within channel configurations.
- APIs to support RAIL timebase synchronization across sleep/wake were added: [RAIL_ConfigSleep\(\)](#), [RAIL_Sleep\(\)](#), and [RAIL_Wake\(\)](#).
- State transitions for success and error are now encapsulated in a new [RAIL_StateTransitions_t](#) structure. This affected the [RAIL_SetRxTransitions\(\)](#) and [RAIL_SetTxTransitions\(\)](#) routines.
- The former [RAIL_EVENT_TX_BUFFER_UNDERFLOW](#) has been broken out into non-ACK ([RAIL_EVENT_TX_UNDERFLOW](#)) and ACK ([RAIL_EVENT_TXACK_UNDERFLOW](#)) flavors like the other transmit completion events.
- Diagnostic toning APIs were merged into the streaming APIs as the [RAIL_StreamMode_t::RAIL_STREAM_CARRIER_WAVE](#).
- [RAIL_ConfigCal\(\)](#) was revamped to use [RAIL_CalMask_t](#).
- [RAIL_Calibrate\(\)](#)'s bool `calSave` argument was removed, being redundant with a NULL vs. non-NULL `calValues` argument.
- Support for Packet Traffic Arbitration (PTA) or Coexistence was added in the form of an API to prevent transmits from going on the air: [RAIL_EnableTxHoldOff\(\)](#), along with [RAIL_IsTxHoldOffEnabled\(\)](#) to check the current state.
- The former [RAIL_AutoAckDisable\(\)](#) was merged into [RAIL_ConfigAutoAck\(\)](#) by means of its [RAIL_AutoAckConfig_t::enable](#) parameter field.
- The former [RAIL_EnableRxFifoThreshold\(\)](#) was replaced by using [RAIL_ConfigEvents\(\)](#) to enable/disable the [RAIL_EVENT_RX_FIFO_ALMOST_FULL](#) event.
- The former blocking [RAIL_PollAverageRssi\(\)](#) was eliminated; use the non-blocking [RAIL_StartAverageRssi\(\)](#) instead.
- Several APIs were renamed for consistency and updated accordingly:
 - former [RAIL_GetTxPacketInfo\(\)](#) is now [RAIL_GetTxPacketDetails\(\)](#);
 - former [RAIL_UseTxBufferForAutoAck\(\)](#) is now [RAIL_UseTxFifoForAutoAck\(\)](#);
 - former [RAIL_LoadAutoAckBuffer\(\)](#) is now [RAIL_WriteAutoAckFifo\(\)](#).
- More API functions were updated to take a [RAIL_Handle_t](#) argument: [RAIL_SetFreqOffset\(\)](#), [RAIL_GetRxFreqOffset\(\)](#), and the [RAIL_IEEE802154_protocol](#) APIs.
- [RAIL_IEEE802154_GetAddress\(\)](#) was updated to return [RAIL_Status_t](#) rather than [RAIL_IEEE802154_Address_t](#) which is now passed as an output argument.
- [RAIL_EnableDirectMode\(\)](#) and [RAIL_SetTune\(\)](#) now return [RAIL_Status_t](#) rather than void.
- A multiprotocol scheduler API was added to get its status: [RAIL_GetSchedulerStatus\(\)](#).
- The [RAILCb_AssertFailed\(\)](#) callback returns with a [RAIL_Handle_t](#) parameter that might be NULL since assert conditions might occur during initialization prior to handle assignment or during multiprotocol switching.
- Many APIs with pointer arguments were clarified to use const pointers where the API doesn't modify the data pointed to.

2.0.0

- All API names were updated to follow a strictly VerbNoun convention.
- [RAIL_SetAbortScheduledTxDuringRx\(\)](#) is no longer a separate API, rather it was added to [RAIL_ScheduleTxConfig_t](#). Set the desired TX during RX behavior when that struct is passed to [RAIL_StartScheduledTx\(\)](#).
- Add support for building single-protocol, multi-protocol, and core RAIL libraries.
- Changed [RAIL_RadioState_t](#) to a bitmask. Supported [RAIL_RadioState_t](#) bits are: [RAIL_RF_STATE_ACTIVE](#), [RAIL_RF_STATE_RX](#), and [RAIL_RF_STATE_TX](#); combinations of these bits yield the additional states [RAIL_RF_STATE_INACTIVE](#), [RAIL_RF_STATE_IDLE](#), [RAIL_RF_STATE_RX_ACTIVE](#), and [RAIL_RF_STATE_TX_ACTIVE](#).
- Added [RAIL_RX_OPTION_TRACK_ABORTED_FRAMES](#) to configure seeing full packets via Packet Trace, even after they have been aborted.
- Added RAIL 15.4 PTI Rx error code `UNDESIRED_ACK` to indicate an ACK filtered because it did not have the desired sequence number matching that in the recently-transmitted ACK-requesting frame.
- Updated return type from `uint8_t` to [RAIL_Status_t](#) for many functions.

- Removed deprecated bit error-rate APIs `RAIL_BerConfigSet()`, `RAIL_BerRxStart()`, `RAIL_BerRxStop()`, `RAIL_BerStatusGet()`, along with deprecated types `RAIL_BerConfig_t` and `RAIL_BerStatus_t`.
 - TX related events are now broken out to specify whether the event is relevant to an ACK packet or user-initiated TX. Specifically those are [RAIL_EVENT_TX_ABORTED](#) and [RAIL_EVENT_TXACK_ABORTED](#), [RAIL_EVENT_TX_BLOCKED](#) and [RAIL_EVENT_TXACK_BLOCKED](#), and [RAIL_EVENT_TX_PACKET_SENT](#) and [RAIL_EVENT_TXACK_PACKET_SENT](#). One of each type (i.e. one ACK, one non-ACK) can occur before being handled.
 - [RAIL_StartTx\(\)](#) no longer takes a pre-TX operation function pointer to do scheduled or CCA transmits. Instead, there are now separate API's for each type of transmit: [RAIL_StartTx\(\)](#), [RAIL_StartScheduledTx\(\)](#), [RAIL_StartCcaLbtTx\(\)](#), [RAIL_StartCcaCsmatx\(\)](#).
 - [RAIL_TxOptions_t](#) has been converted to a 32-bit bitmask, so that RX and TX options are both represented as bitmasks. A single TX option mask is passed into each TX call.
 - [RAIL_ConfigEvents\(\)](#) now also takes two bitmasks, one to indicate which events should be affected, and the second indicating which of those events should be enabled.
 - Similarly, [RAIL_ConfigRxOptions\(\)](#) takes two bitmasks, one to indicate which options to update, and the second indicating the desired value of those options, which affect all future receives.
 - Options for removing RX appended info and ignoring CRC are now controlled exclusively by [RAIL_ConfigRxOptions\(\)](#).
-

1.6.1

- Added new BLE radio configurations for EFR32xG13 parts to fix a transmit overshoot issue on the coded PHY configurations for BLE Long Range.
- Fixed DCDC voltage droop during `RAIL_RfInit()`
- Added an API ([RAIL_EnablePaCal\(\)](#)) that enables loading of chip to chip PA calibration data.

1.6.0

- Fixed an issue where `RAIL_RfStateGet()` would mistakenly return `RAIL_RF_STATE_IDLE` instead of `RAIL_RF_STATE_TX` when a Transmit operation's LBT is active.
 - Added an API (`RAIL_SetAbortScheduledTxDuringRx`) that sets the behavior when a scheduled TX fires during an RX or ACK. If the API is passed false, TX's will be delayed until the RX (or ACK to the RX) is complete. That behavior is the default and what always happened previously. Passing the function true however will abort scheduled TX's that fire during the RX (or ACK to the RX) and fire the `RAILCb_TxRadioStatus` callback.
 - Added new `RAILCb_AssertFailed` callback to give the flexible handling of asserts within RAIL. This includes defines of assert codes and error reasons in [rail_assert_error_codes.h](#). The default implementation hangs in an infinite loop.
 - Added `RAIL_ENUM` define to override enum size ambiguity in the ARM EABI by making them actually a `uint8_t`. For documentation purposes they are still shown as enums.
 - Added new TX options: `removeCrc`, to override whether CRC is sent; and `syncWordId`, to choose which SYNC word to transmit when multiple SYNC words are configured.
 - Added new RX option: `RAIL_RX_OPTION_ENABLE_DUALSYNC`, which allows multiple SYNC words to be searched for on PHYs that support it.
 - Added `RAIL_PeekRxPacket` to allow reading part of a packet before it has been fully received.
 - Changed BER Test functionality in `RAILTest` such that if a RX overflow is detected (due to the incoming datarate being too high), the test now aborts and displays an appropriate status message.
 - Fixed an issue where the code could hang in a RX overflow state when receiving incoming data streams.
 - `RAIL_TxToneStart()` and `RAIL_TxStreamStart()` now idle the radio before transmitting.
 - Fixed an issue where calling `RAIL_TimerSet()` while in the middle of a stream or tone transmission could lock up the radio.
 - Added new frequency offset related functions, `RAIL_GetRxFreqOffset` and `RAIL_SetFreqOffset`, and the associated `RAIL_FrequencyOffset_t` type definition.
-

1.5.2

- Fixed an issue in 1.5.1 that would cause scheduled receive, scheduled transmit, and RAIL timer operations to fail on EFR32xG12 with a relative delay of 0.

- Fixed an issue in 1.5.1 that would cause scheduled receive and scheduled transmit to fail if called with an absolute time from [0, rxWarmTime) or [0, txWarmTime) respectively.
- Fixed an issue where a failed scheduled transmit operation would prevent any transmit from succeeding until RAIL_RfIdle() or RAIL_RfIdleExt() were called.
- Fixed an issue in 1.5.1 where calling idle while looking for an ACK could cause the radio to get stuck in the receive state.

1.5.1

- Added official support for the EFR32xG12 parts through a new librail_efr32xg12.a version of the library.
- Added RAIL_SetRxOptions API for configuration of receive options, such as RAIL_RX_OPTION_STORE_CRC.
- Increase transmit and receive fifo sizes to 512 bytes for all EFR32xG1 parts. This impacts both packet and fifo modes.
- Added [RAIL_RxDataSource_t::RX_DEMOD_DATA](#) in order to receive demodulated data via FIFO method.
- Shortened the duration that the transmit buffer lock in RAIL_TxDataLoad() is held, as it was overprotective. RAIL will still report an error if the application attempts to manipulate the transmit buffer while actively transmitting data.
- Improved state transition timings for enhanced precision and accuracy. To improve precision we have switched to a more stable receive complete event since our previous trigger could jitter for certain radio configurations. To improve accuracy we measured and removed PHY specific overhead for BLE applications. Note that this measurement was BLE specific and does **not** completely correct accuracy on any arbitrary PHY configuration. For custom PHY configurations you will still have to manually calibrate out any offsets by measurement and any previous measurements are likely different now.
- Moved BER test implementation out of the RAIL library and into RAILTest. Deprecated the existing BER test functions in the RAIL API. With BER test code now existing in RAILTest, you must use version 2.2.1 or later of the radio configurator.
- Disabled receive buffer overflow and transmit buffer underflow as BUFC_IRQHandler() events. These events now only occur in FRC_IRQHandler().
- Sped up RAIL transmit and receive buffer read/write operations.
- Fixed a race condition that could cause the receive packet callback to be delayed until the next receive packet event.

1.5.0

- Added other methods of [Data Management](#). The application can configure RAIL to provide/retrieve data via FIFO method.
- The new FIFO method of data management brings with it two new callbacks: RAILCb_TxFifoAlmostEmpty(), and RAILCb_RxFifoAlmostFull(). Note that you must implement these new callbacks or stub them out if you do not intend to use them.

1.4.2

- Fixed a bug in how we handled IEEE 802.15.4 data request packets. Specifically, we were not issuing the RAILCb_IEEE802154_DataRequestCommand() callback for data request packets that were sent with MAC security enabled.
- Fixed a bug where calling RAIL_TxDataLoad() at an inopportune time could cause that and all subsequent such calls to fail.

1.4.1

- Implemented LQI measurement. The field in RAIL_AppendedInfo_t is now populated on every packet receive.
- Allow transmit while receive is active, if the channel is the same. After transmitting, the radio will follow the configured state transitions as before.
- Allow shorter ccaBackoff times if receive is active during the CSMA or LBT process.
- Allow calling RAIL_RxConfig while receiving.
- Added new receive event: RAIL_RX_CONFIG_PACKET_ABORTED
- Added new RAIL_TxConfig API, along with more transmit events: RAIL_TX_CONFIG_CHANNEL_CLEAR, RAIL_TX_CONFIG_CCA_RETRY, RAIL_TX_CONFIG_START_CCA
- Fixed a filtering bug in the IEEE 802.15.4 code. We were incorrectly filtering packets with a 0xFFFF source PANID while the coordinator which caused problems in the joining process. The new logic will only use the source PANID for filtering if we are the coordinator, there is no destination address, and the frame type is MAC command or Data.
- Migrated to Gecko SDK 5.0, including the new em_core APIs for interrupt enabling and disabling. This adds required dependencies on the CORE_EnterCritical() and CORE_ExitCritical() functions.
- Added new RSSI functions for hardware and software averaging: RAIL_StartAverageRSSI(), RAIL_GetAverageRSSI(), RAIL_AverageRSSIReady(), RAIL_PollAverageRSSI().

- Added new RSSI callback: `RAILCb_RssiAverageDone()`.
- Added the ability to switch between multiple BLE phys, including 2Mbps phys, on chips that support the new phys.

1.4.0

- Added support for the EFR32MG12P part family with larger RAM and Flash size variants along with some new chip features.
- Renamed anything that was just efr32 in the previous releases to efr32xg1 to differentiate it from the new efr32xg12 family of parts.
- Add support for the new BLE 2MBit PHY to the BLE acceleration and configuration APIs on EFR32xG12 chips.
- Better optimize IR Calibration run times for all EFR32xG1 parts. You must use version 0.69 or later of the radio configurator.

1.3.2

- Added [RAIL_SetPtiProtocol\(\)](#) and [RAIL_PtiProtocol_t](#). This function allows the application to configure RAIL's packet trace interface to output protocol specific data.
- Fix BER functions to operate at 1Mbps data rates.
- `RAIL_RfIdle()` does not clear the transmit buffer anymore. `RAIL_TxDataLoad()` resets the transmit buffer with every packet set. This fixes the case where `RAIL_RfIdle()` is called after `RAIL_TxDataLoad()`.
- Fixed bug when using fixed packet lengths configured via the Radio Configurator. `RAIL_TxStart()` will not overwrite the configured fixed length to the amount of data loaded by `RAIL_TxDataLoad()`.
- Fixed bug in `RADIO_PA_Init()` where ramp times greater than 11.2 ms would be truncated to a very short ramp.

1.3.1

- Added IEEE 802.15.4 hardware acceleration. The new APIs can be found in [protocol/ieee802154/rail_ieee802154.h](#). Documentation for these APIs can be found in [IEEE 802.15.4](#). This includes a new callback: `RAILCb_IEEE802154_DataRequestCommand()`.
- Added Auto Ack hardware acceleration. Documentation for these APIs can be found in [Auto-Ack](#).
- Added `RAIL_TxStartWithOptions()`. This new API takes a pointer to [RAIL_TxOptions_t](#) that contains options to modify the upcoming transmit.
- Increased minimum state transition time to 100 us.
- Added [RAIL_STATUS_INVALID_CALL](#)
- Added `RAIL_AppendedInfo_t::isAck`

1.3.0

- Added BLE hardware acceleration. The new APIs can be found in [protocol/ble/rail_ble.h](#). Documentation for these APIs can be found in [BLE](#).
- Added [RAIL_ScheduleRx\(\)](#) and [RAIL_ScheduleRxConfig_t](#). This allows the application to schedule a RX window using the RAIL timebase.
- Added [RAIL_TIME_DISABLED](#) enum to [RAIL_TimeMode_t](#).
- Added an extended idle function which gives you more control of how the radio is idled. See `RAIL_RfIdleExt()` and `RAIL_RfIdleMode_t` for more details.
- Added `RAIL_RX_CONFIG_TIMEOUT` and `RAIL_RX_CONFIG_SCHEDULED_RX_END`
- Added `RAIL_TX_CONFIG_TX_ABORTED` and `RAIL_TX_CONFIG_TX_BLOCKED`
- Added an extended radio status callback that supports up to 32 different status reasons (`RAILCb_RxRadioStatusExt()`). You may implement this instead of `RAILCb_RxRadioStatus()` or continue using the old version if you do not need access to more than the first 8 status values.

1.2.7

- Added FEM control signals
- Added `RAIL_SetPaCTune` in order to tune the PA capacitor value for TX and RX.
- Fix typo in `RAIL_StreamMode_t` : `PSUEDO_RANDOM_STREAM` to `PSEUDO_RANDOM_STREAM`

1.2.6

- Added ability to reset demod.

1.2.5

- Fixed register settings for BER testing.
- Fixed packet buffer for contents larger than 170 bytes.

1.2.4

- Presented worst case RfSense wake period

1.2.3

- Optimized address filtering code for operation at higher data rates.
- Clarified data rate limitations of address filtering.

1.2.2

- Added [RAIL_SetTime\(\)](#) in order to allow the user to change the RAIL timebase with microsecond granularity.
- Removed spurious calls to `RAIL_RxRadioStatus()` with the `RAIL_RX_CONFIG_BUFFER_OVERFLOW` argument, which were happening when receive was aborted due to events such as address filtering failures.
- Fixed incorrect PA output power calculations.

1.2.1

- Allow `RAIL_RxStart()` and `RAIL_TxStart()` to not error if the radio is heading to idle after a call to `RAIL_RfIdle()`. Application code does not have to wait until the radio is completely idle before calling a receive or transmit operation as long as the radio is heading to idle.

1.2.0

- Fixed IR calibration such that calibration values are more accurate for radio configurations with datarates below 10Kbps.
- Fixed IR calibration such that the default calibration value is applied when a better calibration value is unable to be calculated.
- Added `RAIL_CallInit()` for calibration initialization.
- Added BER test API and structures for diagnostic use - `RAIL_BerConfigSet()`, `RAIL_BerRxStart()`, `RAIL_BerRxStop()`, `RAIL_BerStatusGet()`, `RAIL_BerConfig_t`, `RAIL_BerStatus_t`.
- Enabled DC Calibration during initialization.

1.1.1

- Fixed CCA timing to prevent invalid CCA failures when starting out of IDLE state.
- Fixed RX state transitions to TX when ignoring CRC errors. Will properly take the success route when a CRC error occurs while ignoring CRC errors.
- Fixed TX error state transition after CCA failure.

1.1.0

- `RAIL_RX_CONFIG_INVALID_CRC` changed to `RAIL_RX_CONFIG_FRAME_ERROR`
 - This now accurately represents the callback flag.
 - A frame error is either an invalid crc, a frame coding error, or a variable length error. On detection of a frame error, the radio will proceed to the defined error state.
- [RAIL_SetRxTransitions\(\)](#) and [RAIL_SetTxTransitions\(\)](#) APIs added
 - Add ability to configure radio to automatically to transition to a given radio state on completion of a receive or transmit event.
- [RAIL_SetStateTiming\(\)](#) API added
 - State transition times can be configured. The given timestamps will not perfectly match on-air times, due to the RX and TX chain delays, which are dependent on the radio configuration.

- If address filtering and frame type length decoding are both enabled, this gives the ability to enable or disable address filtering per frame type

1.0.0

- Return quarter dBm resolution from RAIL_RxGetRSSI(). This changes the return value to an int16_t from an int8_t.
 - Added calibration APIs ([Calibration](#))
 - The RAIL library will notify the application via callback, RAILCb_CalNeeded(), whenever it detects a calibration is needed.
 - An API is provided allowing the application to choose when to calibrate the radio, to pass in a known calibration value, or force a calibration to occur.
 - RAIL_TxPowerSet() returns the power level that can be set which might not match the requested power level.
-

Beta 1 - November 16, 2015

- Exposed EFR32 RF Energy Sensing capability via a RAIL API.
 - This functionality allows the radio to be woken from sleep by a large amount of energy over the air.
- Added new RAIL Timer APIs ([System Timing](#))
 - This provides access to a hardware timer running at the RAIL timebase. You can use this to implement simple protocol timing that may be needed by your low level radio code.
 - This change also adds a new RAILCb_TimerExpired() that must be implemented by your application.
- Added APIs to access the bit rate and symbol rate for the current PHY (RAIL_BitRateGet(), RAIL_SymbolRateGet()).
- Cleaned up RAIL type names so that they all begin with the RAIL_ prefix. This will require updates to existing application code.
- Initial integration with Simplicity Studio's application builder tools

Alpha 2 - September 14, 2015

- Filtering of receive packets based on defined address fields and compare values. See [Address Filtering](#) for more details.
- Added different Pre-Transmit options for the Tx function.
 - Scheduling of packet transmission at an absolute time or after a defined delay.
 - Packet transmission conditional on Clear Channel Assessment (CSMA, LBT).
- The ability to read RSSI manually with RAIL_RxGetRSSI().
- Support for building RAIL applications using both GCC and IAR toolchains.

Alpha 1

- Initial release with support for transmit receive and basic radio configuration.

RAILtest Changelist

RAILtest Changelist

RAIL 2.18.x:

- **1321076:** Fixed an issue with `setDebugSignal` railtest command that broke DOUT and DCLK signals.
- **1419786:** Updated Wi-SUN mode switch channel mask for BZ band in RAILTEST on the EFR32xG25 part.

RAIL 2.16.x:

- **1175616:** Added support for RAILTEST on the MGM240x modules.
- **1185980:** Added support for channel masks during Wi-SUN mode switch in RAILTEST application on the EFR32xG25.

RAIL 2.15.x:

- **1103500:** When `printRxErrors` is in effect, erroneous packets now show RSSI and any erroneous payload received when Rx FIFO mode is used.
- **1105873:** RAILtest now displays board name (if known) on startup.

RAIL 2.13.x:

- **825100:** Added a clearer error message when `setRssiOffset` is used and the radio is not idle.
- **828171:** Added support for all types of line endings to the `sweepTxPower` command to prevent situations where advancing to the next power level could trigger twice in a row.
- **845292:** Added `resetCause` (in hex) to railtest's startup display.

RAIL 2.12.x:

- **721006:** Any pin on ports A, B, C, or D can now be selected to use as a debug signal in railtest using the `setDebugSignal` command.

RAIL 2.11.x:

- **679427:** Altered RAILtest to allow for more dependencies to be removed - antenna diversity, RF path, CLI, CLI RAM storage, CLI delay.
- **689827:** Fixed a hanging issue caused by Rx channel hopping being enabled without a successful Rx channel hopping configuration. This was previously a recommended practice but is now required.
- **700831:** RAILtest sleep command again allows distinguishing energy modes 4h and 4s.
- **712345:** Added additional startup messages to RAILtest showing radio, system, and PTI configuration information.
- **737886:** Fixed the help text for the `setTimings` command which had the order swapped for the rxToTx and txToRx time parameters.

RAIL 2.10.x:

- **605067:** The `SL_RAIL_TEST_PER` warning has been removed from RAILtest when a PER pin is not configured for use.

RAIL 2.9.x:

- **381485:** Added `configChannelHoppingMulti` command for pre-configuring channel hopping multi-sense mode parameters prior to issuing `configRxChannelHopping` command. Also augmented `configRxDutyCycle` command to allow parameterization of this mode.
- **402203:** Augmented `configChannelHoppingOptions` command to allow specifying RSSI threshold when enabling that option. Augmented `configRxDutyCycle` command to allow specifying options in general including the RSSI threshold.

- **427841:** Updated the Command Line Interface (CLI) help display format.
- **488425:** CLI commands are now displayed back to the user using the same capitalization as they are entered (e.g. entering command `configTXoptions` will display `configTXoptions` and not `configTxOptions`).
- **454890:** Restructured RAILtest to use a common main.c template with a software component-based architecture to select software features.
- **487098:** Disabled graphics, buttons, LEDs and PER test GPIO pin by default.
- **484333:** Standardized spelling of `RAILtest` for consistency.
 - Updated bootup message example: `{{(reset)}}{App:RAILtest}{Built:Jul 19 2019 13:36:24}}`
- **633846:** Altered CLI input of command `setDebugSignal` .
- **630457:** Fixed an issue where the RAIL PTI component would automatically reserve hardware pins when no pins were specified for use.

RAIL 2.8.x:

- **344361:** Added new command `getCteSampleRate` to retrieve the actual ADC sample rate used to capture the CTE.
- **369727:** Added new commands `setTxTimePos` and `setRxTimePos` to designate the time position of displayed packet timestamps among the `RAIL_PacketTimePosition_t` choices. In txEnd messages, now also display the lastTxStart time, and fixed a bug with the lastTxStatus display.
- **406293:** Fixed an issue where a memory pointer could be discarded from the event queue on queue overflow and never released. This can occur if the memory buffer pool is greater than the event queue size.
- **415017:** Updated the RAILTest bootup message to incorporate the `reset` command, the application name and the date and time of the RAILTest build.
 - Updated bootup message example: `{{(reset)}}{App:RAIL Test App}{Built:Jul 19 2019 13:36:24}}`
- **406080:** Added new commands `configRfSenseWakeupPhy` and `setRfSenseTxPayload` to support waking up chips that support RFSENSE Selective(OOK) mode. Updated `sleep` and `rfsense` commands to allow setting the syncword length and syncword for EFR32XG22 chips which support RFSENSE Selective(OOK) mode. The `tx` command will always overwrite the TX FIFO with a default payload, hence `setRfSenseTxPayload` must be used with `fifoModeTestOptions` set to use `txFifoManual` to prevent the FIFO being overwritten.
- **460516:** Fixed minor railtest problem where `setConfigIndex` might not restart receiver listening like `setChannel` would if the receiver was on when the command was issued.

RAIL 2.7.x:

- **368510:** Fixed an issue if receive was entered because of a transmit state transition where we would not change the channel when calling `setChannel` and would ignore calls to `rx 1` to enter normal receive mode.
- **394431:** Fixed an issue where the configured antenna diversity settings in HAL config would not be applied to the radio at startup.

RAIL 2.6.x:

- Fixed `setMultiTimer` which was ignoring its mode argument; `abs` mode now works.
- The commands `setRxOptions` and `configTxOptions` can now be called without any parameters to print the current state of these options.
- RAILTest now uses its config-changed callback for both external and internal radio configs to ensure that the appropriate PA level is always selected.
- Added 802.15.4 `setDataReqLatency` command for testing purposes.
- Fixed some asynchronous event print outs related to Z-Wave beam frame reception since they could corrupt the output.
- Added `getChannelHoppingRssi` command to read the RSSI when in channel hopping mode on each available channel.

RAIL 2.5.x:

- Added the following multitimer commands.
 - `enableMultiTimer` , `setMultiTimer` , `multiTimerCancel` , `printMultiTimerStats`
- Added `printRxErrors` command to enable/disable printing of unsuccessfully received packets.
- Added the ability to save command scripts to flash and run them from flash. Renamed command `startScript` to `runScript` . Added command `clearScript` .

RAIL 2.4.x:

- Added the RAM Modem Reconfiguration commands. These commands allow the dynamic configuration of the modem via a set of studio scripted commands.

RAIL 2.0.x:

- `setChannel` no longer requires the radio be idle.

RAIL 1.6.x:

- simple `tx` command transmits with defaults (i.e. don't wait for ack, send CRC, use sync word 0)
- `configTxOptions` now controls 3 options
 - BIT0: Wait for ack
 - BIT1: Send CRC
 - BIT2: Sync Word ID
- `setRxOptions` now controls 3 options
 - BIT0: Store CRC
 - BIT1: Ignore CRC errors
 - BIT2: Enable dual sync

RAIL 1.5.x:

- Added the following commands:
 - `rxConfig`, `setRxOptions`, `setFixedLength`, `fifoStatus`, `dataConfig`, `setTxFifoThreshold`, `setRxFifoThreshold`, `iqCapture`

RAIL 1.4.1:

- Added the following commands:
 - `setPtiProtocol`, `sweepPower`, `startAvgRssi`, `getAvgRssi`, `setBle1Mbps`, `setBle2Mbps`

RAIL 1.3.1:

- Added Bluetooth Low Energy commands
 - `setBleMode`, `getBleMode`, `setBleChannelParams`, `setBleAdvertising`
- Fixed help text for `setPanId802154`, `setShortAddr802154`, `setLongAddr802154`
- Changed parameters for `acceptFrames` separating the bitmask into individual elements

RAIL 1.3.0:

- Added Auto Acknowledgement commands
 - `autoAckConfig`, `autoAckDisable`, `setAckPayload`, `setAckLength`, `printAckPacket`, `getAutoAck`, `autoAckPause`, `setTxAckOptions`
- Added IEEE 802.15.4 commands
 - `enable802154`, `config2p4GHz802154`, `acceptFrames`, `setPromiscuousMode`, `setPanCoordinator`, `setPanId802154`, `setShortAddr802154`, `setLongAddr802154`
- Added scheduled RX command `rxAt`
- Added `transmitWithOptions` and `configTxOptions` commands
- Added `setDebugSignal` command.

RAIL 1.2.1:

- Added `setPaCtune` which can configure custom PA capacitor tuning values for TX and RX
- Fixed RSSI slope requiring a 4dB power change for 1dB change in the received packet print

RAIL 1.2.0:

- Added bit error rate test commands - `setBerConfig`, `berRx`, and `berStatus`.
- Added packet error rate test commands - `perRx`, `perStatus`.
- Added the ability to set the RAIL timebase in microseconds - `setTime`.

- RxOverflow counter now only counts number of overflows, instead of mistakenly counting some non-overflow events.
- Removed the RfUs output parameter from the 'sleep' command due to it being incorrect.

RAIL 1.1 - May 15, 2016:

- Automatic state transitions and timings added via `setRxTransitions`, `setTxTransitions`, and `setTimings`.
- Ability to selectively enable the address filter on certain frame types added through `addressFilterByFrame` command.
- Independent status booleans were removed in favor of a single `AppMode` status variable, which keeps track of the state of RAILtest.
- More validation is done to ensure that certain RAIL commands are only run in the correct `AppMode` and radio state. The radio must be IDLE for `setChannel`, `freqOverride`, `setPower`, and `setCtune` to succeed.
- `setTestMode` was removed in favor of individual commands for `setTxUnderflow`, `setRxOverflow`, and `setCalibrations`.
- `toggleTone` became `setTxTone`, and `toggleStream` became `setTxStream`.
- Removed command `getTicks`, since the `RAIL_GetTime()` function made the timer used unnecessary.
- Removed command `txTimeReport`, since it was exposing implementation-specific behavior encapsulated by RAIL.

Beta 1 - November 16, 2015:

- New library features for RF Energy Sensing exposed via new `rfSense` and `sleep` commands.
- New RAIL timer functionality exposed via the `setTimer`, `timerCancel`, and `printTimerStats` commands.
- New `printDataRates` command to get information about the current PHY's data rates.
- Documentation clean up.

Alpha 2 - September 14, 2015:

- Commands for filtering of receive packets based on defined address fields and compare values.
- Scheduling of packet transmission at an absolute time or after a defined delay.
- Packet transmission conditional on Clear Channel Assessment (CSMA, LBT).
- The ability to read RSSI manually.
- Capability to switch between multiple PHY configurations at runtime.
- Revised help text (more uniform parameter descriptions).

API Changes

API Changes

RAIL Library 2.19.0

Added

New API Name	Return Value	Comment
sl_rail_util_pa_nvm_read_config(sl_rail_nvm_pa_config_t *p_pa_config)	sl_status_t	new API that reads stored sl_rail_nvm_pa_config_t from NVM
sl_rail_util_pa_nvm_read_mode(void)	RAIL_TxPowerMode_t	new API that reads stored

[RAIL_TxPowerMode_t](#) from NVM | | [sl_rail_util_pa_nvm_write_config\(const sl_rail_nvm_pa_config_t *p_pa_config\)](#) | [sl_status_t](#) | new API that stores a [sl_rail_nvm_pa_config_t](#) into NVM | | [sl_rail_util_pa_nvm_write_mode\(RAIL_TxPowerMode_t pa_mode\)](#) | [sl_status_t](#) | new API that stores a [RAIL_TxPowerMode_t](#) into NVM |

Modified

Old API Name	Return Value	New API Name	Return Value
RAIL_ZWAVE_ConfigBeamRx(RAIL_Handle_t railHandle, RAIL_ZWAVE_BeamRxConfig_t *config)	RAIL_Status_t	RAIL_ZWAVE_ConfigBeamRx(RAIL_Handle_t railHandle, const RAIL_ZWAVE_BeamRxConfig_t *config)	RAIL_Status_t
RAIL_ZWAVE_SetTxLowPowerDbm(RAIL_Handle_t railHandle, RAIL_TxPower_t powerLevel)	RAIL_Status_t	RAIL_ZWAVE_SetTxLowPowerDbm(RAIL_Handle_t railHandle, RAIL_TxPower_t power)	RAIL_Status_t

RAIL Library 2.18.2

No changes

RAIL Library 2.18.1

Added

New API Name	Return Value	Comment
RAIL_ConvertDbmToPowerSettingEntry(RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPower_t power, RAIL_TxPowerSettingEntry_t *pPowerSettingInfo)	RAIL_Status_t	API to convert desired decibel value to a RAIL_TxPowerSettingEntry_t
RAIL_TriggerRxChannelHop(RAIL_Handle_t railHandle)	RAIL_Status_t	API to trigger RX channel hopping manually

RAIL Library 2.18.0

Added

New API Name	Return Value	Comment
<code>RAIL_ConfigChannelsAlt(RAIL_Handle_t railHandle, const RAIL_ChannelConfig_t *config, RAIL_RadioConfigChangedCallback_t cb)</code>	<code>RAIL_Status_t</code>	new API to just configure the channels but not tune the radio to the first channel
<code>RAIL_GetRxIncomingPacketRssi(RAIL_Handle_t railHandle)</code>	<code>int8_t</code>	new API to return the RSSI associated with the incoming packet
<code>RAIL_SupportsTrustZoneSecurePeripherals(RAIL_Handle_t railHandle)</code>	<code>bool</code>	new API to be used by trustzone secured application to indicate whether the chip supports TrustZone secure configuration of peripherals used by RAIL
<code>RAIL_TZ_CheckPeripheralsSecureStates(void)</code>	<code>RAIL_Status_t</code>	new API to be used by trustzone secured application to check the secure state of peripherals used by RAIL
<code>RAIL_TZ_ConfigAntennaGpio(const RAIL_AntennaConfig_t *config)</code>	<code>RAIL_Status_t</code>	new API to be used by trustzone secured application to set GPIO for antenna config
<code>RAIL_TZ_ConfigureHfxo(void)</code>	<code>RAIL_Status_t</code>	new API to be used by trustzone secured application to configure HFXO
<code>RAIL_TZ_DisableSecureRadioIrqs(void)</code>	<code>RAIL_Status_t</code>	new API to be used by trustzone secured application to disable secure peripheral interrupts needed by the radio
<code>RAIL_TZ_EnableSecureRadioIrqs(void)</code>	<code>RAIL_Status_t</code>	new API to be used by trustzone secured application to enable secure peripheral interrupts needed by the radio
<code>RAIL_TZ_InitNonSecure(const RAIL_TZ_Config_t *pTzConfig)</code>	<code>RAIL_Status_t</code>	new API to be used by trustzone secured application to initialize RAIL TrustZone feature for non-secure world
<code>RAIL_TZ_InitSecure(void)</code>	<code>RAIL_Status_t</code>	new API to be used by trustzone secured application to initialize RAIL TrustZone feature for secure world
<code>RAIL_TZ_RadioClockEnable(void)</code>	<code>RAIL_Status_t</code>	new API to be used by trustzone secured application to enable radio clocks
<code>RAIL_TZ_RadioPerformM2mLdma(uint32_t *pDest, const uint32_t *pSrc, uint32_t numWords)</code>	<code>RAIL_Status_t</code>	new API to be used by trustzone secured application to perform LDMA transfer for the radio
<code>RAIL_TZ_ReadInternalTemperature(uint16_t *internalTemperatureKelvin, bool enableTemperatureInterrupts)</code>	<code>RAIL_Status_t</code>	new API to be used by trustzone secured application to read the internal temperature
<code>RAIL_TZ_RfecaClockEnable(void)</code>	<code>RAIL_Status_t</code>	new API to be used by trustzone secured application to enable RFECA clocks
<code>RAIL_TZ_RfecalsClockEnabled(void)</code>	<code>bool</code>	new API to be used by trustzone secured application to tell whether RFECA clocks are enabled

Modified

Old API Name	Return Value	New API Name	Return Value	Comment
RAIL_EnableAutoLnaBypass(RAIL_Handle_t railHandle, bool enable, const RAIL_AutoLnaBypassConfig_t *pAutoLnaBypassConfig)	RAIL_Status_t	RAIL_EnablePrsLnaBypass(RAIL_Handle_t railHandle, bool enable, const RAIL_PrsLnaBypassConfig_t *pPrsLnaBypassConfig)	RAIL_Status_t	API RAIL_Enable replaced by RAIL_Enable
RAIL_SupportsAutoLnaBypass(RAIL_Handle_t railHandle)	bool	RAIL_SupportsPrsLnaBypass(RAIL_Handle_t railHandle)	bool	API RAIL_Supports is replaced by RAIL_Supports

RAIL Library 2.17.2

No changes

RAIL Library 2.17.1

Added

New API Name	Return Value	Comment
RAIL_EnableAutoLnaBypass(RAIL_Handle_t railHandle, bool enable, const RAIL_AutoLnaBypassConfig_t *pAutoLnaBypassConfig)	RAIL_Status_t	API to enable automatic LNA bypass for external FEM
RAIL_SupportsAutoLnaBypass(RAIL_Handle_t railHandle)	bool	API to indicate whether this chip supports automatic LNA bypass for external FEM.

Modified

Old API Name	Return Value	New API Name	Return Value	Comment
RAIL_BLE_PhySwitchToRx(RAIL_Handle_t railHandle, RAIL_BLE_Phy_t phy, uint16_t railChannel, uint32_t startRxTime, uint32_t crclnit, uint32_t accessAddress, uint16_t logicalChannel, bool disableWhitening)	RAIL_Status_t	RAIL_BLE_PhySwitchToRx(RAIL_Handle_t railHandle, RAIL_BLE_Phy_t phy, uint16_t railChannel, RAIL_Time_t startRxTime, uint32_t crclnit, uint32_t accessAddress, uint16_t logicalChannel, bool disableWhitening)	RAIL_Status_t	Changed the data of the startRxTime uint32_t to RAIL_Time_t
RAIL_ConfigSleepAlt(RAIL_Handle_t railHandle, RAIL_TimerSyncConfig_t *syncConfig)	RAIL_Status_t	RAIL_ConfigSleepAlt(RAIL_Handle_t railHandle, const RAIL_TimerSyncConfig_t *syncConfig)	RAIL_Status_t	Added const to RAIL_TimerSyncConfig_t *syncConfig
RAIL_Init(RAIL_Config_t *railCfg, RAIL_InitCompleteCallbackPtr_t cb)	RAIL_Handle_t	RAIL_Init(const RAIL_Config_t *railCfg, RAIL_InitCompleteCallbackPtr_t cb)	RAIL_Handle_t	Added const to RAIL_Config_t *railCfg

RAIL Library 2.17.0

Added

New API Name	Return Value	Comment
RAIL_ConfigVdet(RAIL_Handle_t genericRailHandle, const RAIL_VdetConfig_t *config)	RAIL_Status_t	API to configure VDET plugin
RAIL_EnableVdet(RAIL_Handle_t genericRailHandle, bool enable)	RAIL_Status_t	API to enable VDET plugin

New API Name	Return Value	Comment
RAIL_GetVdet(RAIL_Handle_t genericRailHandle, uint32_t *pVdetMv)	RAIL_Status_t	API to get the VDET measurement
RAIL_GetVdetConfig(RAIL_Handle_t genericRailHandle, RAIL_VdetConfig_t *config)	RAIL_Status_t	API to get the VDET plugin configuration
RAIL_IsVdetEnabled(RAIL_Handle_t genericRailHandle)	bool	API to check if VDET is enabled
RAIL_SupportsPathDiversity(RAIL_Handle_t railHandle)	bool	API to indicate if the chip supports internal RF path diversity
RAIL_SupportsVdet(RAIL_Handle_t railHandle)	bool	API to indicate if the chip supports VDET

Removed

Old API Name	Return Value	Comment
RAIL_ConfigEff(RAIL_Handle_t genericRailHandle, const RAIL_EffConfig_t *config)	RAIL_Status_t	API removed
RAIL_GetSetEffClpcControl(RAIL_Handle_t railHandle, uint16_t tempBuffer[(52U)/sizeof(uint16_t)], bool reset)	RAIL_Status_t	API removed
RAIL_GetSetEffClpcEnable(RAIL_Handle_t railHandle, uint8_t *newClpcEnable, bool changeClpcEnable)	RAIL_Status_t	API removed
RAIL_GetSetEffClpcFastLoop(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, uint16_t *newTargetMv, uint16_t *newSlopeMvPerPaLevel, bool changeValues)	RAIL_Status_t	API removed
RAIL_GetSetEffClpcFastLoopCal(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, RAIL_EffCalConfig_t *calibrationEntry, bool changeValues)	RAIL_Status_t	API removed
RAIL_GetSetEffClpcFastLoopCalSlp(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, int16_t *newSlope1e1MvPerDdbm, int16_t *newoffset290Ddbm, bool changeValues)	RAIL_Status_t	API removed
RAIL_GetSetEffClpcFemdata(RAIL_Handle_t railHandle, uint8_t *newMode, bool changeMode)	RAIL_Status_t	API removed
RAIL_GetSetEffLnaBypassDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)	RAIL_Status_t	API removed
RAIL_GetSetEffLnaRuralUrbanMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)	RAIL_Status_t	API removed
RAIL_GetSetEffLnaUrbanBypassMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)	RAIL_Status_t	API removed
RAIL_GetSetEffLnaUrbanDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)	RAIL_Status_t	API removed
RAIL_GetSetEffTempThreshold(RAIL_Handle_t railHandle, uint16_t *newThresholdK, bool changeThreshold)	RAIL_Status_t	API removed
RAIL_SupportsEff(RAIL_Handle_t railHandle)	bool	API removed

Modified

Old API Name	Return Value	New API Name	Return Value	Comments
RAIL_BLE_Deinit(RAIL_Handle_t railHandle)	void	RAIL_BLE_Deinit(RAIL_Handle_t railHandle)	RAIL_Status_t	Modified API to return RAIL_Status_t instead of void
RAIL_BLE_Init(RAIL_Handle_t railHandle)	void	RAIL_BLE_Init(RAIL_Handle_t railHandle)	RAIL_Status_t	Modified API to return RAIL_Status_t instead of void
RAIL_CancelTimer(RAIL_Handle_t railHandle)	void	RAIL_CancelTimer(RAIL_Handle_t railHandle)	RAIL_Status_t	Modified API to return RAIL_Status_t instead of void
RAIL_EnableTxHoldOff(RAIL_Handle_t railHandle, bool enable)	void	RAIL_EnableTxHoldOff(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	Modified API to return RAIL_Status_t instead of void
RAIL_GetRxIncomingPacketInfo(RAIL_Handle_t railHandle, RAIL_RxPacketInfo_t *pPacketInfo)	void	RAIL_GetRxIncomingPacketInfo(RAIL_Handle_t railHandle, RAIL_RxPacketInfo_t *pPacketInfo)	RAIL_Status_t	Modified API to return RAIL_Status_t instead of void
RAIL_GetTemperature(RAIL_Handle_t railHandle, int16_t tempBuffer[((3U)+(6U)+(2U)+(1U))], bool reset)	RAIL_Status_t	RAIL_GetTemperature(RAIL_Handle_t railHandle, int16_t tempBuffer[((3U)+(1U))], bool reset)	RAIL_Status_t	Updated size of array in tempBuffer
RAIL_GetVersion(RAIL_Version_t *version, bool verbose)	void	RAIL_GetVersion(RAIL_Version_t *version, bool verbose)	RAIL_Status_t	Modified API to return RAIL_Status_t instead of void
RAIL_IEEE802154_WriteEnhAck(RAIL_Handle_t railHandle, const uint8_t *ackData, uint8_t ackDataLen)	RAIL_Status_t	RAIL_IEEE802154_WriteEnhAck(RAIL_Handle_t railHandle, const uint8_t *ackData, uint16_t ackDataLen)	RAIL_Status_t	Updated API to take uint16_t ackDataLen parameter instead of uint8_t ackDataLen parameter
RAIL_Idle(RAIL_Handle_t railHandle, RAIL_IdleMode_t mode, bool wait)	void	RAIL_Idle(RAIL_Handle_t railHandle, RAIL_IdleMode_t mode, bool wait)	RAIL_Status_t	Modified API to return RAIL_Status_t instead of void

Old API Name	Return Value	New API Name	Return Value	Comments
RAIL_IncludeFrameTypeLength(RAIL_Handle_t railHandle)	void	RAIL_IncludeFrameTypeLength(RAIL_Handle_t railHandle)	RAIL_Status_t	Modified API to return RAIL_Status_t instead of void
RAIL_PauseRxAutoAck(RAIL_Handle_t railHandle, bool pause)	void	RAIL_PauseRxAutoAck(RAIL_Handle_t railHandle, bool pause)	RAIL_Status_t	Modified API to return RAIL_Status_t instead of void
RAIL_PauseTxAutoAck(RAIL_Handle_t railHandle, bool pause)	void	RAIL_PauseTxAutoAck(RAIL_Handle_t railHandle, bool pause)	RAIL_Status_t	Modified API to return RAIL_Status_t instead of void
RAIL_ResetAddressFilter(RAIL_Handle_t railHandle)	void	RAIL_ResetAddressFilter(RAIL_Handle_t railHandle)	RAIL_Status_t	Modified API to return RAIL_Status_t instead of void
RAIL_ResetFifo(RAIL_Handle_t railHandle, bool txFifo, bool rxFifo)	void	RAIL_ResetFifo(RAIL_Handle_t railHandle, bool txFifo, bool rxFifo)	RAIL_Status_t	Modified API to return RAIL_Status_t instead of void
RAIL_VerifyTxPowerCurves(const struct RAIL_TxPowerCurvesConfigAlt *config)	void	RAIL_VerifyTxPowerCurves(const struct RAIL_TxPowerCurvesConfigAlt *config)	RAIL_Status_t	Modified API to return RAIL_Status_t instead of void
RAIL_WriteAutoAckFifo(RAIL_Handle_t railHandle, const uint8_t *ackData, uint8_t ackDataLen)	RAIL_Status_t	RAIL_WriteAutoAckFifo(RAIL_Handle_t railHandle, const uint8_t *ackData, uint16_t ackDataLen)	RAIL_Status_t	Updated API to take uint16_t ackDataLen parameter instead of uint8_t ackDataLen parameter
RAIL_YieldRadio(RAIL_Handle_t railHandle)	void	RAIL_YieldRadio(RAIL_Handle_t railHandle)	RAIL_Status_t	Modified API to return RAIL_Status_t instead of void

RAIL Library 2.16.2

No changes

RAIL Library 2.16.1

Added

New API Name	Return Value	Comment
RAIL_GetCrcInitVal(RAIL_Handle_t railHandle)	uint32_t	A new API to get the CRC initialization value
RAIL_GetWhiteningInitVal(RAIL_Handle_t railHandle)	uint16_t	A new API to get the whitening initialization value
RAIL_ResetCrcInitVal(RAIL_Handle_t railHandle)	RAIL_Status_t	A new API to restore the CRC initialization value to its initial setting from the Radio Configurator
RAIL_ResetWhiteningInitVal(RAIL_Handle_t railHandle)	RAIL_Status_t	A new API to restore the whitening initialization value to its initial setting from the Radio Configurator
RAIL_SetCrcInitVal(RAIL_Handle_t railHandle, uint32_t crcInit)	RAIL_Status_t	A new API to set the CRC initialization value
RAIL_SetWhiteningInitVal(RAIL_Handle_t railHandle, uint16_t whiteInit)	RAIL_Status_t	A new API to set the whitening initialization value

RAIL Library 2.16.0

Added

New API Name	Return Value	Comment
RAILCb_IEEE802154_IsModeSwitchNewChannelValid(uint32_t currentBaseFreq, uint8_t newPhyModelId, const RAIL_ChannelConfigEntry_t *configEntryNewPhyModelId, uint16_t *pChannel)	RAIL_Status_t	A new API to check if the mode switch channel is valid
RAIL_SupportsCollisionDetection(RAIL_Handle_t railHandle)	bool	A new API to indicate if the chip supports Collision detection feature

RAIL Library 2.15.2

Added

New API Name	Return Value	Comment
RAIL_SupportsTxRepeatStartToStart(RAIL_Handle_t railHandle)	bool	See RAIL Changelist 1131398

RAIL Library 2.15.1

Added

New API Name	Return Value	Comment
RAIL_GetAutoAckFifo(RAIL_Handle_t railHandle, uint8_t **ackBuffer, uint16_t *ackBufferBytes)	RAIL_Status_t	See RAIL Changelist 748817
RAIL_Sidewalk_ConfigPhy2GFSK50kbps(RAIL_Handle_t railHandle)	RAIL_Status_t	Function to load the Sidewalk 2GFSK 50kbps PHY
RAIL_SupportsProtocolSidewalk(RAIL_Handle_t railHandle)	bool	API to determine at runtime when Sidewalk PHYs are supported

Modified

Old API Name	Return Value	New API Name	Return Value	Comment
RAIL_ConfigVerification(RAIL_Handle_t railHandle, RAIL_VerifyConfig_t *configVerify, const uint32_t *radioConfig, RAIL_VerifyCallbackPtr_t cb)	RAIL_Status_t	RAIL_ConfigVerification(RAIL_Handle_t railHandle, RAIL_VerifyConfig_t *configVerify, RAIL_RadioConfig_t radioConfig, RAIL_VerifyCallbackPtr_t cb)	RAIL_Status_t	

RAIL Library 2.15.0

Added

New API Name	Return Value	Comment
RAIL_EnableCacheSynthCal(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	See RAIL Changelist item 1061665.
RAIL_GetSetEffClpcFemdata(RAIL_Handle_t railHandle, uint8_t *newMode, bool changeMode)	RAIL_Status_t	New alpha API.
RAIL_IEEE802154_SetRxToEnhAckTx(RAIL_Handle_t railHandle, RAIL_TransitionTime_t *pRxToEnhAckTx)	RAIL_Status_t	See RAIL Changelist item 1117127.
RAIL_SetTxFifoAlt(RAIL_Handle_t railHandle, uint8_t *addr, uint16_t startOffset, uint16_t initLength, uint16_t size)	uint16_t	See RAIL Changelist item 1103587.
RAIL_SupportsFastRx2Rx(RAIL_Handle_t railHandle)	bool	Added to help determine Fast RX2RX support at runtime.
RAIL_SupportsProtocolWiSUN(RAIL_Handle_t railHandle)	bool	Added to help determine WiSUN support at runtime.
RAIL_WMBUS_Config(RAIL_Handle_t railHandle, bool enableSimultaneousTCRx)	RAIL_Status_t	See RAIL Changelist item 1099829.
RAIL_WMBUS_SupportsSimultaneousTCRx(RAIL_Handle_t railHandle)	bool	See RAIL Changelist item 1099829.

Removed

Old API Name	Return Value	Comment
RAIL_GetSetEffMode(RAIL_Handle_t railHandle, uint8_t *newMode, bool changeMode)	RAIL_Status_t	Removed alpha API.

Modified

Old API Name	Return Value	New API Name	Return Value	Comment
RAIL_GetSetClpcEnable(RAIL_Handle_t railHandle, uint8_t *newClpcEnable, bool changeClpcEnable)	RAIL_Status_t	RAIL_GetSetEffClpcEnable(RAIL_Handle_t railHandle, uint8_t *newClpcEnable, bool changeClpcEnable)	RAIL_Status_t	Renamed alpha API.
RAIL_GetSetClpcFastLoop(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, uint16_t *newTargetMv, uint16_t *newSlopeMvPerPaLevel, bool changeValues)	RAIL_Status_t	RAIL_GetSetEffClpcFastLoop(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, uint16_t *newTargetMv, uint16_t *newSlopeMvPerPaLevel, bool changeValues)	RAIL_Status_t	Renamed alpha API.

Old API Name	Return Value	New API Name	Return Va
RAIL_GetSetClpcFastLoopCal(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, RAIL_EffCalConfig_t *calibrationEntry, bool changeValues)	RAIL_Status_t	RAIL_GetSetEffClpcFastLoopCal(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, RAIL_EffCalConfig_t *calibrationEntry, bool changeValues)	RAIL_Stat
RAIL_GetSetClpcFastLoopCalSlp(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, int16_t *newSlope1e1MvPerDdbm, int16_t *newoffset290Ddbm, bool changeValues)	RAIL_Status_t	RAIL_GetSetEffClpcFastLoopCalSlp(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, int16_t *newSlope1e1MvPerDdbm, int16_t *newoffset290Ddbm, bool changeValues)	RAIL_Stat
RAIL_GetSetEffBypassDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)	RAIL_Status_t	RAIL_GetSetEffLnaBypassDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)	RAIL_Stat
RAIL_GetSetEffControl(RAIL_Handle_t railHandle, uint16_t tempBuffer[(52U)/sizeof(uint16_t)], bool reset)	RAIL_Status_t	RAIL_GetSetEffClpcControl(RAIL_Handle_t railHandle, uint16_t tempBuffer[(52U)/sizeof(uint16_t)], bool reset)	RAIL_Stat
RAIL_GetSetEffRuralUrbanMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)	RAIL_Status_t	RAIL_GetSetEffLnaRuralUrbanMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)	RAIL_Stat
RAIL_GetSetEffUrbanBypassMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)	RAIL_Status_t	RAIL_GetSetEffLnaUrbanBypassMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)	RAIL_Stat
RAIL_GetSetEffUrbanDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)	RAIL_Status_t	RAIL_GetSetEffLnaUrbanDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)	RAIL_Stat

RAIL Library 2.14.3

No changes

RAIL Library 2.14.2

Added

New API Name	Return Value	Comment
RAIL_GetPowerSettingTable(RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPower_t *minPower, RAIL_TxPower_t *maxPower, RAIL_TxPowerLevel_t *step)	const RAIL_PaPowerSetting_t	
RAIL_ZWAVE_GetBeamHomeldHash(RAIL_Handle_t railHandle, RAIL_ZWAVE_HomeldHash_t *pBeamHomeldHash)	RAIL_Status_t	

Removed

Old API Name	Return Value	Comment
RAIL_GetFemProtectionConfig(RAIL_Handle_t railHandle, RAIL_FemProtectionConfig_t *femConfig)	RAIL_Status_t	Removed alpha API for EFR32xG25+EFF01 support
RAIL_SetFemProtectionConfig(RAIL_Handle_t railHandle, const RAIL_FemProtectionConfig_t *newFemConfig)	RAIL_Status_t	Removed alpha API for EFR32xG25+EFF01 support

RAIL Library 2.14.1

No changes

RAIL Library 2.14.0

Added

New API Name	Return Value	Comment
RAIL_CalibrateHFXO(RAIL_Handle_t railHandle, int8_t *crystalPPMError)	RAIL_Status_t	See RAIL Changelist item 751910
RAIL_CompensateHFXO(RAIL_Handle_t railHandle, int8_t crystalPPMError)	RAIL_Status_t	See RAIL Changelist item 751910
RAIL_ConfigHFXOCompensation(RAIL_Handle_t railHandle, const RAIL_HFXOCompensationConfig_t *pHfxoCompensationConfig)	RAIL_Status_t	See RAIL Changelist item 751910
RAIL_ConfigThermalProtection(RAIL_Handle_t genericRailHandle, const RAIL_ChipTempConfig_t *chipTempConfig)	RAIL_Status_t	See RAIL Changelist item 1022182
RAIL_GetHFXOCompensationConfig(RAIL_Handle_t railHandle, RAIL_HFXOCompensationConfig_t *pHfxoCompensationConfig)	RAIL_Status_t	See RAIL Changelist item 751910
RAIL_GetPaPowerSetting(RAIL_Handle_t railHandle)	RAIL_PaPowerSetting_t	
RAIL_GetSetClpcFastLoopCal(RAIL_Handle_t railHandle, RAIL_EffCalConfigEnum_t calibrationIndex, RAIL_EffCalConfig_t *calibrationEntry, bool changeValues)	RAIL_Status_t	
RAIL_GetSetClpcFastLoopCalSlp(RAIL_Handle_t railHandle, RAIL_EffCalConfigEnum_t calibrationIndex, int16_t *newSlope1e1, int16_t *newoffset290Ddbm, bool changeValues)	RAIL_Status_t	
RAIL_GetThermalProtection(RAIL_Handle_t genericRailHandle, RAIL_ChipTempConfig_t *chipTempConfig)	RAIL_Status_t	See RAIL Changelist item 1022182
RAIL_IEEE802154_ConfigRxChannelSwitching(RAIL_Handle_t railHandle, const RAIL_IEEE802154_RxChannelSwitchingCfg_t *pConfig)	RAIL_Status_t	See RAIL Changelist item 858567
RAIL_IEEE802154_SupportsRxChannelSwitching(RAIL_Handle_t railHandle)	bool	See RAIL Changelist item 858567
RAIL_SetPaPowerSetting(RAIL_Handle_t railHandle, RAIL_PaPowerSetting_t paPowerSetting, RAIL_TxPower_t minPowerDbm, RAIL_TxPower_t maxPowerDbm, RAIL_TxPower_t currentPowerDbm)	RAIL_Status_t	
RAIL_SupportsHFXOCompensation(RAIL_Handle_t railHandle)	bool	See RAIL Changelist item 751910
RAIL_SupportsIEEE802154Band2P4(RAIL_Handle_t railHandle)	bool	

New API Name	Return Value	Comment
RAIL_SupportsThermalProtection(RAIL_Handle_t railHandle)	bool	See RAIL Changelist item 1022182
RAIL_ZWAVE_PerformIrcal(RAIL_Handle_t railHandle, RAIL_ZWAVE_IrcalVal_t *plrCalVals, bool forceIrcal)	RAIL_Status_t	See RAIL Changelist item 846240

Removed

Old API Name	Return Value	Comment
RAIL_GetSetClpcSlowLoop(RAIL_Handle_t railHandle, uint16_t *newTarget, uint16_t *newSlope, bool changeValues)	RAIL_Status_t	Removed alpha API for EFR32xG25+EFF01 support
RAIL_GetSetInternalTempThreshold(RAIL_Handle_t railHandle, uint16_t *newThreshold, bool changeThreshold)	RAIL_Status_t	Removed alpha API for EFR32xG25 support

Modified

Old API Name	Return Value	New API Name	Return Value	Comment
RAIL_GetSetClpcFastLoop(RAIL_Handle_t railHandle, uint16_t *newTarget, uint16_t *newSlope, bool changeValues)	RAIL_Status_t	RAIL_GetSetClpcFastLoop(RAIL_Handle_t railHandle, uint16_t *newTargetFsk, uint16_t *newSlopeFsk, uint16_t *newTargetOfdm, uint16_t *newSlopeOfdm, bool changeValues)	RAIL_Status_t	Modified alpha API

RAIL Library 2.13.3

No changes

RAIL Library 2.13.2

No changes

RAIL Library 2.13.1

No changes

RAIL Library 2.13.0

Added

New API Name	Return Value	Comment
RAIL_ChangedDcdc(void)	RAIL_Status_t	
RAIL_ComputeHFXOPPMError(RAIL_Handle_t railHandle, int16_t crystalTemperatureC, int8_t *crystalPPMError)	RAIL_Status_t	
RAIL_ConfigEff(RAIL_Handle_t railHandle, const RAIL_EffConfig_t *config)	RAIL_Status_t	
RAIL_ConfigHFXOThermistor(RAIL_Handle_t railHandle, const RAIL_HFXOThermistorConfig_t *hfxoThermistorConfig)	RAIL_Status_t	
RAIL_ConfigPaAutoEntry(RAIL_Handle_t railHandle, const RAIL_PaAutoModeConfigEntry_t *paAutoModeEntry)	RAIL_Status_t	
RAIL_ConvertThermistorImpedance(RAIL_Handle_t railHandle, uint32_t thermistorImpedance, int16_t *thermistorTemperatureC)	RAIL_Status_t	

New API Name	Return Value	Comment
RAIL_GetChannelAlt(RAIL_Handle_t railHandle, uint16_t *channel)	RAIL_Status_t	
RAIL_GetFemProtectionConfig(RAIL_Handle_t railHandle, RAIL_FemProtectionConfig_t *femConfig)	RAIL_Status_t	
RAIL_GetRssiDetectThreshold(RAIL_Handle_t railHandle)	int8_t	
RAIL_GetSetClpcEnable(RAIL_Handle_t railHandle, uint8_t *newClpcEnable, bool changeClpcEnable)	RAIL_Status_t	
RAIL_GetSetClpcFastLoop(RAIL_Handle_t railHandle, uint16_t *newTarget, uint16_t *newSlope, bool changeValues)	RAIL_Status_t	
RAIL_GetSetClpcSlowLoop(RAIL_Handle_t railHandle, uint16_t *newTarget, uint16_t *newSlope, bool changeValues)	RAIL_Status_t	
RAIL_GetSetEffBypassDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)	RAIL_Status_t	
RAIL_GetSetEffControl(RAIL_Handle_t railHandle, uint16_t tempBuffer[(52U)/sizeof(uint16_t)], bool reset)	RAIL_Status_t	
RAIL_GetSetEffMode(RAIL_Handle_t railHandle, uint8_t *newMode, bool changeMode)	RAIL_Status_t	
RAIL_GetSetEffRuralUrbanMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)	RAIL_Status_t	
RAIL_GetSetEffTempThreshold(RAIL_Handle_t railHandle, uint16_t *newThreshold, bool changeThreshold)	RAIL_Status_t	
RAIL_GetSetEffUrbanBypassMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)	RAIL_Status_t	
RAIL_GetSetEffUrbanDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)	RAIL_Status_t	
RAIL_GetSetInternalTempThreshold(RAIL_Handle_t railHandle, uint16_t *newThreshold, bool changeThreshold)	RAIL_Status_t	
RAIL_GetTemperature(RAIL_Handle_t railHandle, int16_t tempBuffer[(13U)], bool reset)	RAIL_Status_t	
RAIL_SetFemProtectionConfig(RAIL_Handle_t railHandle, const RAIL_FemProtectionConfig_t *newFemConfig)	RAIL_Status_t	
RAIL_SetRssiDetectThreshold(RAIL_Handle_t railHandle, int8_t rssiThresholdDbm)	RAIL_Status_t	
RAIL_SupportsAuxAdc(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsEff(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsRssiDetectThreshold(RAIL_Handle_t railHandle)	bool	

Removed

Old API Name	Return Value	Comment
RAIL_IEEE802154_Config863MHzSUNOFDMOptRadio(RAIL_Handle_t railHandle, RAIL_IEEE802154_OFDMOption_t ofdmOption)	RAIL_Status_t	Removed unnecessary alpha API.
RAIL_IEEE802154_Config902MHzSUNOFDMOptRadio(RAIL_Handle_t railHandle, RAIL_IEEE802154_OFDMOption_t ofdmOption)	RAIL_Status_t	Removed unnecessary alpha API.

Modified

Old API Name	Return Value	New API Name	Return Value
RAIL_BLE_ConfigSignalIdentifier(RAIL_Handle_t railHandle, const RAIL_BLE_SignalIdentifierMode_t signalIdentifierMode)	RAIL_Status_t	RAIL_BLE_ConfigSignalIdentifier(RAIL_Handle_t railHandle, RAIL_BLE_SignalIdentifierMode_t signalIdentifierMode)	RAIL_Status_t
RAIL_BLE_EnableSignalIdentifier(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	RAIL_BLE_EnableSignalDetection(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t
RAIL_IEEE802154_ConfigCcaMode(RAIL_Handle_t railHandle, const RAIL_IEEE802154_CcaMode_t ccaMode)	RAIL_Status_t	RAIL_IEEE802154_ConfigCcaMode(RAIL_Handle_t railHandle, RAIL_IEEE802154_CcaMode_t ccaMode)	RAIL_Status_t
RAIL_IEEE802154_ConfigSignalIdentifier(RAIL_Handle_t railHandle)	RAIL_Status_t	RAIL_IEEE802154_ConfigSignalIdentifier(RAIL_Handle_t railHandle, RAIL_IEEE802154_SignalIdentifierMode_t signalIdentifierMode)	RAIL_Status_t
RAIL_IEEE802154_EnableSignalIdentifier(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	RAIL_IEEE802154_EnableSignalDetection(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t

RAIL Library 2.12.2

No changes

RAIL Library 2.12.1

No changes

RAIL Library 2.12.0

Added

New API Name	Return Value	Comment
RAIL_AddStateBuffer3(RAIL_Handle_t genericRailHandle)	RAIL_Status_t	See RAIL Changelist item 721163
RAIL_AddStateBuffer4(RAIL_Handle_t genericRailHandle)	RAIL_Status_t	See RAIL Changelist item 721163

New API Name	Return Value	Comment
RAIL_BLE_ConfigSignalIdentifier(RAIL_Handle_t railHandle, const RAIL_BLE_SignalIdentifierMode_t signalIdentifierMode)	RAIL_Status_t	
RAIL_BLE_EnableSignalIdentifier(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	
RAIL_BLE_GetCteSampleOffset(RAIL_Handle_t railHandle)	uint8_t	
RAIL_BLE_SupportsSignalIdentifier(RAIL_Handle_t railHandle)	bool	
RAIL_GetSchedulerStatusAlt(RAIL_Handle_t railHandle, RAIL_SchedulerStatus_t *pSchedulerStatus, RAIL_Status_t *pRailStatus)	RAIL_Status_t	See RAIL Changelog item 726491
RAIL_GetTxPacketsRemaining(RAIL_Handle_t railHandle)	uint16_t	See RAIL Changelog item 708206
RAIL_IEEE802154_ComputeChannelFromPhyModelId(RAIL_Handle_t railHandle, uint8_t newPhyModelId, uint16_t *pChannel)	RAIL_Status_t	
RAIL_IEEE802154_Config2p4GHzRadioCustom1(RAIL_Handle_t railHandle)	RAIL_Status_t	See RAIL Changelog item 727728
RAIL_IEEE802154_Config863MHzSUNOFDMOptRadio(RAIL_Handle_t railHandle, RAIL_IEEE802154_OFDMOption_t ofdmOption)	RAIL_Status_t	
RAIL_IEEE802154_Config902MHzSUNOFDMOptRadio(RAIL_Handle_t railHandle, RAIL_IEEE802154_OFDMOption_t ofdmOption)	RAIL_Status_t	
RAIL_IEEE802154_ConfigCcaMode(RAIL_Handle_t railHandle, const RAIL_IEEE802154_CcaMode_t ccaMode)	RAIL_Status_t	
RAIL_IEEE802154_ConfigSignalIdentifier(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IEEE802154_EnableSignalIdentifier(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	
RAIL_IEEE802154_SupportsCustom1Phy(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsDualPaConfig(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsGDynFec(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsGModeSwitch(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsSignalIdentifier(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsOFDMPA(RAIL_Handle_t railHandle)	bool	

RAIL Library 2.11.3

No changes

RAIL Library 2.11.2

Added

New API Name	Return Value	Comment
RAIL_ConfigDirectMode(RAIL_Handle_t railHandle, const RAIL_DirectModeConfig_t *directModeConfig)	RAIL_Status_t	
RAIL_EnableDirectModeAlt(RAIL_Handle_t railHandle, bool enableDirectTx, bool enableDirectRx)	RAIL_Status_t	
RAIL_GetDefaultRxDutyCycleConfig(RAIL_Handle_t railHandle, RAIL_RxDutyCycleConfig_t *config)	RAIL_Status_t	

New API Name	Return Value	Comment
RAIL_SetMfmPingPongFifo(RAIL_Handle_t railHandle, const RAIL_MFM_PingPongBufferConfig_t *config)	RAIL_Status_t	
RAIL_SupportsMfm(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsRxDirectModeDataToFifo(RAIL_Handle_t railHandle)	bool	

RAIL Library 2.11.1

Added

New API Name	Return Value	Comment
RAIL_GetPtiProtocol(RAIL_Handle_t railHandle)	RAIL_PtiProtocol_t	
RAIL_GetRadioClockFreqHz(RAIL_Handle_t railHandle)	uint32_t	

RAIL Library 2.11.0

Added

New API Name	Return Value	Comment
RAIL_BLE_SetNextTxRepeat(RAIL_Handle_t railHandle, const RAIL_BLE_TxRepeatConfig_t *repeatConfig)	RAIL_Status_t	
RAIL_SetNextTxRepeat(RAIL_Handle_t railHandle, const RAIL_TxRepeatConfig_t *repeatConfig)	RAIL_Status_t	
RAIL_StartScheduledCcaCsmaTx(RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_ScheduleTxConfig_t *scheduleTxConfig, const RAIL_CsmaConfig_t *csmaConfig, const RAIL_SchedulerInfo_t *schedulerInfo)	RAIL_Status_t	
RAIL_StartScheduledCcaLbtTx(RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_ScheduleTxConfig_t *scheduleTxConfig, const RAIL_LbtConfig_t *lbtConfig, const RAIL_SchedulerInfo_t *schedulerInfo)	RAIL_Status_t	
RAIL_SupportsTxToTx(RAIL_Handle_t railHandle)	bool	
RAIL_ZWAVE_GetBeamRssi(RAIL_Handle_t railHandle, int8_t *pBeamRssi)	RAIL_Status_t	
RAIL_ZWAVE_GetRxBeamConfig(RAIL_ZWAVE_BeamRxConfig_t *pConfig)	RAIL_Status_t	
RAIL_ZWAVE_SetDefaultRxBeamConfig(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_ZWAVE_SetLrAckData(RAIL_Handle_t railHandle, const RAIL_ZWAVE_LrAckData_t *pLrAckData)	RAIL_Status_t	

RAIL Library 2.10.2

Added

New API Name	Return Value	Comment
RAIL_IEEE802154_GetPtiRadioConfig(RAIL_Handle_t railHandle)	RAIL_IEEE802154_PtiRadioConfig_t	

RAIL Library 2.10.1

No changes

RAIL Library 2.10.0

Added

New API Name	Return Value	Comment
RAIL_ApplyIrcalibrationAlt(RAIL_Handle_t railHandle, RAIL_IrCalValues_t *imageRejection, RAIL_AntennaSel_t rfPath)	RAIL_Status_t	
RAIL_BLE_ConfigPhySimulscan(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_BLE_SupportsSimulscanPhy(RAIL_Handle_t railHandle)	bool	
RAIL_CalibrateIrcalAlt(RAIL_Handle_t railHandle, RAIL_IrCalValues_t *imageRejection, RAIL_AntennaSel_t rfPath)	RAIL_Status_t	
RAIL_ConfigRetimeOptions(RAIL_Handle_t railHandle, RAIL_RetimeOptions_t mask, RAIL_RetimeOptions_t options)	RAIL_Status_t	
RAIL_ConfigSleepAlt(RAIL_Handle_t railHandle, RAIL_TimerSyncConfig_t *syncConfig)	RAIL_Status_t	
RAIL_GetRadioStateDetail(RAIL_Handle_t railHandle)	RAIL_RadioStateDetail_t	
RAIL_GetRetimeOptions(RAIL_Handle_t railHandle, RAIL_RetimeOptions_t *pOptions)	RAIL_Status_t	
RAIL_GetRfPath(RAIL_Handle_t railHandle, RAIL_AntennaSel_t *rfPath)	RAIL_Status_t	
RAIL_GetRssiAlt(RAIL_Handle_t railHandle, RAIL_Time_t waitTimeout)	int16_t	
RAIL_GetTuneDelta(RAIL_Handle_t railHandle)	int32_t	
RAIL_IEEE802154_Config2p4GHzRadioAntDivCoexFem(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IEEE802154_Config2p4GHzRadioAntDivFem(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IEEE802154_Config2p4GHzRadioCoexFem(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IEEE802154_Config2p4GHzRadioFem(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IEEE802154_SupportsFemPhy(RAIL_Handle_t railHandle)	bool	
RAIL_SetAddressFilterAddressMask(RAIL_Handle_t railHandle, uint8_t field, const uint8_t *bitMask)	RAIL_Status_t	
RAIL_SetTuneDelta(RAIL_Handle_t railHandle, int32_t delta)	RAIL_Status_t	
RAIL_SupportsAddrFilterAddressBitMask(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsAddrFilterMask(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsDirectMode(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsRfSenseEnergyDetection(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsSQPhy(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsTxPowerModeAlt(RAIL_Handle_t railHandle, RAIL_TxPowerMode_t *powerMode, RAIL_TxPowerLevel_t *maxPowerLevel, RAIL_TxPowerLevel_t *minPowerLevel)	bool	
RAIL_ZWAVE_ConfigRxChannelHopping(RAIL_Handle_t railHandle, RAIL_RxChannelHoppingConfig_t *config)	RAIL_Status_t	
RAIL_ZWAVE_GetLrBeamTxPower(RAIL_Handle_t railHandle, uint8_t *pLrBeamTxPower)	RAIL_Status_t	
RAIL_ZWAVE_GetRegion(RAIL_Handle_t railHandle)	RAIL_ZWAVE_RegionId_t	
RAIL_ZWAVE_SupportsConcPhy(RAIL_Handle_t railHandle)	bool	
RAIL_ZWAVE_SupportsEnergyDetectPhy(RAIL_Handle_t railHandle)	bool	

RAIL Library 2.9.2

No changes

RAIL Library 2.9.1

No changes

RAIL Library 2.9.0

Added

New API Name	Return Value	Comment
RAILCb_ConfigSleepTimerSync(RAIL_TimerSyncConfig_t *timerSyncConfig)	void	
RAIL_BLE_ConfigPhyQuuppa(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_BLE_SupportsQuuppa(RAIL_Handle_t railHandle)	bool	
RAIL_DeinitPowerManager(void)	RAIL_Status_t	
RAIL_GetTxPacketDetailsAlt2(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_GetTxTimeFrameEndAlt(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_GetTxTimePreambleStartAlt(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_GetTxTimeSyncWordEndAlt(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_InitPowerManager(void)	RAIL_Status_t	

RAIL Library 2.8.10

No changes

RAIL Library 2.8.9

No changes

RAIL Library 2.8.8

No changes

RAIL Library 2.8.7

No changes

RAIL Library 2.8.6

No changes

RAIL Library 2.8.4

Added

New API Name	Return Value	Comment
RAIL_BLE_ConfigAoxAntenna(RAIL_Handle_t railHandle, RAIL_BLE_AoxAntennaConfig_t *antennaConfig)	RAIL_Status_t	
RAIL_BLE_InitCte(RAIL_Handle_t railHandle)	RAIL_Status_t	

RAIL Library 2.8.3

Added

New API Name	Return Value	Comment
RAIL_BLE_Supports1Mbps(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_Supports1MbpsNonViterbi(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_Supports1MbpsViterbi(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_Supports2Mbps(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_Supports2MbpsNonViterbi(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_Supports2MbpsViterbi(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_SupportsAntennaSwitching(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_SupportsCodedPhy(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_SupportsCte(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_SupportsIQSampling(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_SupportsPhySwitchToRx(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsCancelFramePendingLookup(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsCoexPhy(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsEEEnhancedAck(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsEMultipurposeFrames(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsESubsetGB868(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsEarlyFramePendingLookup(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsG4ByteCrc(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsGSubsetGB868(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsGUnwhitenedRx(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsGUnwhitenedTx(RAIL_Handle_t railHandle)	bool	
RAIL_Supports2p4GHzBand(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsAlternateTxPower(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsAntennaDiversity(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsChannelHopping(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsDualBand(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsDualSyncWords(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsExternalThermistor(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsPrecisionLFRCO(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsProtocolBLE(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsProtocolIEEE802154(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsProtocolZWave(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsRadioEntropy(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsRfSenseSelectiveOok(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsSubGHzBand(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsTxPowerMode(RAIL_Handle_t railHandle, RAIL_TxPowerMode_t powerMode, RAIL_TxPowerLevel_t *pMaxPowerLevel)	bool	
RAIL_ZWAVE_SupportsRegionPti(RAIL_Handle_t railHandle)	bool	

RAIL Library 2.8.2

No changes

RAIL Library 2.8.1

Added

New API Name	Return Value	Comment
RAILCb_PaAutoModeDecision (RAIL_Handle_t railHandle, RAIL_TxPower_t *power, RAIL_TxPowerMode_t *mode, const RAIL_ChannelConfigEntry_t *chCfgEntry)	RAIL_Status_t	
RAIL_GetThermistorImpedance (RAIL_Handle_t railHandle, uint32_t *thermistorImpedance)	RAIL_Status_t	
RAIL_IEEE802154_ConvertRssiToEd (int8_t rssiDbm)	uint8_t	
RAIL_IEEE802154_ConvertRssiToLqi (uint8_t origLqi, int8_t rssiDbm)	uint8_t	
RAIL_StartThermistorMeasurement (RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_StartTxStreamAlt (RAIL_Handle_t railHandle, uint16_t channel, RAIL_StreamMode_t mode, RAIL_TxOptions_t options)	RAIL_Status_t	

RAIL Library 2.8.0

Added

New API Name	Return Value	Comment
RAIL_BLE_GetCteSampleRate (RAIL_Handle_t railHandle)	uint32_t	
RAIL_ConfigRfSenseSelectiveOokWakeupPhy (RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_ConfigSyncWords (RAIL_Handle_t railHandle, const RAIL_SyncWordConfig_t *syncWordConfig)	RAIL_Status_t	
RAIL_EnablePaAutoMode (RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	
RAIL_GetChannelMetadata (RAIL_Handle_t railHandle, RAIL_ChannelMetadata_t *channelMetadata, uint16_t *length, uint16_t minChannel, uint16_t maxChannel)	RAIL_Status_t	
RAIL_GetRxTransitions (RAIL_Handle_t railHandle, RAIL_StateTransitions_t *transitions)	RAIL_Status_t	
RAIL_GetSyncWords (RAIL_Handle_t railHandle, RAIL_SyncWordConfig_t *syncWordConfig)	RAIL_Status_t	
RAIL_GetTxTransitions (RAIL_Handle_t railHandle, RAIL_StateTransitions_t *transitions)	RAIL_Status_t	
RAIL_IsPaAutoModeEnabled (RAIL_Handle_t railHandle)	bool	
RAIL_SetRfSenseSelectiveOokWakeupPayload (RAIL_Handle_t railHandle, uint8_t preamble, uint8_t numSyncwordBytes, uint32_t syncword)	RAIL_Status_t	
RAIL_StartSelectiveOokRfSense (RAIL_Handle_t railHandle, RAIL_RfSenseSelectiveOokConfig_t *config)	RAIL_Status_t	
RAIL_StopInfinitePreambleTx (RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_ZWAVE_ConfigBeamRx (RAIL_Handle_t railHandle, RAIL_ZWAVE_BeamRxConfig_t *config)	RAIL_Status_t	
RAIL_ZWAVE_ReceiveBeam (RAIL_Handle_t railHandle, uint8_t *beamDetectIndex, const RAIL_SchedulerInfo_t *schedulerInfo)	RAIL_Status_t	

Modified

Old API Name	Return Value	New API Name	Return Value	Comment
RAIL_ZWAVE_ConfigRegion(RAIL_Handle_t railHandle, const RAIL_ZWAVE_RegionConfig_t regionCfg)	RAIL_Status_t	RAIL_ZWAVE_ConfigRegion(RAIL_Handle_t railHandle, const RAIL_ZWAVE_RegionConfig_t *regionCfg)	RAIL_Status_t	

RAIL Library 2.7.4

No changes

RAIL Library 2.7.3

No changes

RAIL Library 2.7.2

No changes

RAIL Library 2.7.1

No changes

RAIL Library 2.7.0

Added

New API Name	Return Value	Comment
RAIL_GetRxIncomingPacketInfo(RAIL_Handle_t railHandle, RAIL_RxPacketInfo_t *pPacketInfo)	void	
RAIL_IEEE802154_ConfigEOptions(RAIL_Handle_t railHandle, RAIL_IEEE802154_EOptions_t mask, RAIL_IEEE802154_EOptions_t options)	RAIL_Status_t	
RAIL_IEEE802154_ConfigGOptions(RAIL_Handle_t railHandle, RAIL_IEEE802154_GOptions_t mask, RAIL_IEEE802154_GOptions_t options)	RAIL_Status_t	
RAIL_IEEE802154_EnableDataFramePending(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	
RAIL_IEEE802154_EnableEarlyFramePending(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	
RAIL_IEEE802154_WriteEnhAck(RAIL_Handle_t railHandle, const uint8_t *ackData, uint8_t ackDataLen)	RAIL_Status_t	
RAIL_UseDma(uint8_t channel)	RAIL_Status_t	
RAIL_ZWAVE_GetTxLowPower(RAIL_Handle_t railHandle)	RAIL_TxPowerLevel_t	
RAIL_ZWAVE_GetTxLowPowerDbm(RAIL_Handle_t railHandle)	RAIL_TxPower_t	
RAIL_ZWAVE_SetTxLowPower(RAIL_Handle_t railHandle, uint8_t powerLevel)	RAIL_Status_t	
RAIL_ZWAVE_SetTxLowPowerDbm(RAIL_Handle_t railHandle, RAIL_TxPower_t powerLevel)	RAIL_Status_t	

RAIL Library 2.6.5

No changes

RAIL Library 2.6.4

No changes

RAIL Library 2.6.3

No changes

RAIL Library 2.6.2

No changes

RAIL Library 2.6.1

Added

New API Name	Return Value	Comment
RAIL_GetChannelHoppingRssi(RAIL_Handle_t railHandle, uint8_t channelIndex)	int16_t	

RAIL Library 2.6.0

Added

New API Name	Return Value	Comment
RAILCb_ConfigFrameTypeLength(RAIL_Handle_t railHandle, const RAIL_FrameType_t *frameType)	void	
RAIL_BLE_ConfigAox(RAIL_Handle_t railHandle, const RAIL_BLE_AoxConfig_t *aoxConfig)	RAIL_Status_t	
RAIL_BLE_CteBufferIsLocked(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_LockCteBuffer(RAIL_Handle_t railHandle, bool lock)	bool	
RAIL_BLE_PhySwitchToRx(RAIL_Handle_t railHandle, RAIL_BLE_Phy_t phy, uint16_t railChannel, uint32_t startRxTime, uint32_t crclnit, uint32_t accessAddress, uint16_t logicalChannel, bool disableWhitening)	RAIL_Status_t	
RAIL_ConfigRxDutyCycle(RAIL_Handle_t railHandle, const RAIL_RxDutyCycleConfig_t *config)	RAIL_Status_t	
RAIL_DelayUs(RAIL_Time_t microseconds)	RAIL_Status_t	
RAIL_EnableRxDutyCycle(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	
RAIL_GetRxTimeFrameEndAlt(RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_GetRxTimePreambleStartAlt(RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_GetRxTimeSyncWordEndAlt(RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_GetTransitionTime(void)	RAIL_Time_t	
RAIL_IEEE802154_Config2p4GHzRadioAntDivCoex(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IEEE802154_Config2p4GHzRadioCoex(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IncludeFrameTypeLength(RAIL_Handle_t railHandle)	void	
RAIL_IsInitialized(void)	bool	
RAIL_SetTransitionTime(RAIL_Time_t transitionTime)	void	
RAIL_ZWAVE_GetBeamChannelIndex(RAIL_Handle_t railHandle, uint8_t *pChannelIndex)	RAIL_Status_t	

RAIL Library 2.5.1

No changes

RAIL Library 2.5.0

Added

New API Name	Return Value	Comment
New API Name	Return Value	Comment
RAIL_ConfigRxChannelHopping(RAIL_Handle_t railHandle, RAIL_RxChannelHoppingConfig_t *config)	RAIL_Status_t	
RAIL_ConvertLqi(RAIL_Handle_t railHandle, RAIL_ConvertLqiCallback_t cb)	RAIL_Status_t	
RAIL_EnableRxChannelHopping(RAIL_Handle_t railHandle, bool enable, bool reset)	RAIL_Status_t	
RAIL_IEEE802154_Config2p4GHzRadioAntDiv(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_SetTxAltPreambleLength(RAIL_Handle_t railHandle, uint16_t length)	RAIL_Status_t	
RAIL_VerifyTxPowerCurves(const struct RAIL_TxPowerCurvesConfigAlt *config)	void	
RAIL_ZWAVE_ConfigOptions(RAIL_Handle_t railHandle, RAIL_ZWAVE_Options_t mask, RAIL_ZWAVE_Options_t options)	RAIL_Status_t	
RAIL_ZWAVE_ConfigRegion(RAIL_Handle_t railHandle, const RAIL_ZWAVE_RegionConfig_t regionCfg)	RAIL_Status_t	
RAIL_ZWAVE_Deinit(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_ZWAVE_GetBeamNodeId(RAIL_Handle_t railHandle, RAIL_ZWAVE_NodeId_t *pNodeId)	RAIL_Status_t	
RAIL_ZWAVE_Init(RAIL_Handle_t railHandle, const RAIL_ZWAVE_Config_t *config)	RAIL_Status_t	
RAIL_ZWAVE_IsEnabled(RAIL_Handle_t railHandle)	bool	
RAIL_ZWAVE_SetHomeld(RAIL_Handle_t railHandle, RAIL_ZWAVE_Homeld_t homeld, RAIL_ZWAVE_HomeldHash_t homeldHash)	RAIL_Status_t	
RAIL_ZWAVE_SetNodeId(RAIL_Handle_t railHandle, RAIL_ZWAVE_NodeId_t nodeId)	RAIL_Status_t	

RAIL Library 2.4.1

No changes

RAIL Library 2.4.0

Added

New API Name	Return Value	Comment
PRORTC_IRQHandler(void)	void	
RAILCb_SetupRxFifo(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_BLE_CalibrateIr(RAIL_Handle_t railHandle, uint32_t *imageRejection)	RAIL_Status_t	
RAIL_GetChannel(RAIL_Handle_t railHandle, uint16_t *channel)	RAIL_Status_t	
RAIL_GetRssiOffset(RAIL_Handle_t railHandle)	int8_t	
RAIL_GetRxPacketDetailsAlt(RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, RAIL_RxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_GetRxTimeFrameEnd(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)	RAIL_Status_t	
RAIL_GetRxTimePreambleStart(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)	RAIL_Status_t	
RAIL_GetRxTimeSyncWordEnd(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)	RAIL_Status_t	
RAIL_GetTxPacketDetailsAlt(RAIL_Handle_t railHandle, bool isAck, RAIL_Time_t *pPacketTime)	RAIL_Status_t	

New API Name	Return Value	Comment
RAIL_GetTxTimeFrameEnd(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)	RAIL_Status_t	
RAIL_GetTxTimePreambleStart(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)	RAIL_Status_t	
RAIL_GetTxTimeSyncWordEnd(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)	RAIL_Status_t	
RAIL_IEEE802154_CalibrateIr2p4Ghz(RAIL_Handle_t railHandle, uint32_t *imageRejection)	RAIL_Status_t	
RAIL_IEEE802154_CalibrateIrSubGhz(RAIL_Handle_t railHandle, uint32_t *imageRejection)	RAIL_Status_t	
RAIL_IEEE802154_ConfigGB863MHzRadio(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IEEE802154_ConfigGB915MHzRadio(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_PrepareChannel(RAIL_Handle_t railHandle, uint16_t channel)	RAIL_Status_t	
RAIL_SetRssiOffset(RAIL_Handle_t railHandle, int8_t rssiOffset)	RAIL_Status_t	
RAIL_SetRxFifo(RAIL_Handle_t railHandle, uint8_t *addr, uint16_t *size)	RAIL_Status_t	
RAIL_SetTaskPriority(RAIL_Handle_t railHandle, uint8_t priority, RAIL_TaskType_t taskType)	RAIL_Status_t	
RAIL_StopTx(RAIL_Handle_t railHandle, RAIL_StopMode_t mode)	RAIL_Status_t	

Multi-Protocol Transition Time Utility

Multi-Protocol Transition Time Utility

The **Multi-Protocol Transition Time Utility** is a software component designed to dynamically determine and configure the appropriate transition time for switching between protocols in a multi-protocol environment. This ensures proper operation of the radio for a given system clock frequency, and maximizes protocol access to the shared radio.

Overview

The **Multi-Protocol Transition Time Utility** is designed to determine and configure the worst-case transition times required for switching between protocols in a dynamic multiprotocol (DMP) application. These transition times represent how early the radio scheduler must begin a protocol switch before a scheduled event in order to not miss it.

The default configuration is tuned specifically for BLE and Zigbee use cases, ensuring reliable operation across Silicon Labs RAIL stacks. Customers can further customize these transition times to optimize performance for their specific applications.

This utility uses a configurable table of frequency thresholds and corresponding transition times to dynamically select the optimal transition time based on the current system clock frequency. By doing so, it ensures efficient protocol switching while maintaining reliable radio performance.

As the system clock frequency increases, the performance of the protocol switch process improves. For example, at 150 MHz, the radio can switch from one protocol to another more quickly than at 38.4 MHz, enabling faster transitions and better overall access to the shared radio in high-frequency systems.

Key Features

- **Dynamic Transition Time Selection:** Automatically selects the appropriate transition time based on the current system clock frequency.
- **Configurable Table:** Users can define custom frequency-to-transition-time mappings using the override functionality.
- **Default Compatibility:** Pre-configured to work with all Silicon Labs stacks on supported part families.
- **Override Support:** Customers can override the default configuration to optimize for their specific device family or application requirements.

Configuration Options

Default Configuration

The default configuration is defined in the `sl_rail_util_mp_transition_time` component. It includes a table of frequency thresholds and corresponding transition times, which are used to determine the appropriate transition time for the current system clock frequency.

Example default configuration for device family sixx301 in `.slcc`:

```
template_contribution:
- name: sl_rail_util_mp_transition_time_table
  value:
    minimum_frequency_hz: "150000000, 38400000, 20000000, 10000000"
    transition_time_us: "350, 450, 1000, 2000"
  condition: [device_generic_family_sixx301]
  unless: [sl_rail_util_mp_transition_time_override]
```

Override Functionality

Customers can override the default values by requiring the `sl_rail_util_mp_transition_time_override` component in their SLCP file. This allows them to define their own `sl_rail_util_mp_transition_time_table` using the same format as the default configuration.

Example override configuration in `.slcp` :

```
requires:
- name: sl_rail_util_mp_transition_time_override
template_contribution:
- name: sl_rail_util_mp_transition_time_table
  value:
    minimum_frequency_hz: "100000000, 50000000"
    transition_time_us: "400, 800"
```

By using the override functionality, customers can fine-tune the transition times to better suit their specific device family or application requirements.

How It Works

1. **Initialization:** The utility initializes by determining the current system clock frequency and selecting the appropriate transition time from the table.
2. **Dynamic Selection:** During runtime, the utility dynamically selects the transition time based on the system clock frequency, ensuring proper protocol transitions for the radio. It is only run once during startup, however the APIs are available for subsequent usage or testing.
3. **RAIL Configuration:** The selected transition time is passed to the RAIL library `sl_rail_set_transition_time` API.

API Reference

`sl_rail_util_mp_transition_time_get`

```
sl_status_t sl_rail_util_mp_transition_time_get(
  const sl_rail_util_mp_transition_time_table_t *p_table,
  const size_t table_size,
  sl_rail_time_t *p_transition_time
);
```

Retrieves the appropriate transition time based on the current system clock frequency.

Parameters:

- `p_table` : Pointer to the transition time table.
- `table_size` : Number of entries in the table.
- `p_transition_time` : Pointer to store the resulting transition time (in microseconds).

Returns:

- `SL_STATUS_OK` : Transition time successfully retrieved.

- `SL_STATUS_INVALID_PARAMETER` : Invalid input parameters.
- `SL_STATUS_NOT_FOUND` : No matching entry found for the current system clock frequency.

<tt>sl_rail_util_mp_transition_time_init</tt>

```
void sl_rail_util_mp_transition_time_init(void);
```

Initializes the transition time utility by determining the appropriate transition time and configuring the RAIL library.

Deprecated List

Deprecated List

- Global [ant0Loc](#)
Backwards compatible name for [RAIL_AntennaConfig_t::defaultPath](#) field.
- Global [RAIL_AntennaConfig_t::ant1Loc](#)
No longer used (ignored).
- Global [RAIL_ApplyIrCalibration](#) (RAIL_Handle_t railHandle, uint32_t imageRejection)
Please use [RAIL_ApplyIrCalibrationAlt](#) instead.
- Global [RAIL_BLE_AOX_OPTIONS_DO_SWITCH](#)
Obsolete AOX option
- Global [RAIL_BLE_AOX_OPTIONS_LOCK_CTE_BUFFER_SHIFT](#)
Please use [RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK_SHIFT](#) instead.
- Global [RAIL_BLE_AOX_OPTIONS_RX_ENABLED](#)
Obsolete AOX option
- Global [RAIL_BLE_AOX_OPTIONS_TX_ENABLED](#)
Obsolete AOX option
- Global [RAIL_BLE_Coding_125kbps_DSA](#)
Will be removed in a future version of RAIL.
- Global [RAIL_BLE_Coding_500kbps_DSA](#)
Will be removed in a future version of RAIL.
- Global [RAIL_BLE_ConfigPhy1Mbps](#) (RAIL_Handle_t railHandle)
BLE non-Viterbi PHYs are no longer supported.
- Global [RAIL_BLE_ConfigPhy2Mbps](#) (RAIL_Handle_t railHandle)
BLE non-Viterbi PHYs are no longer supported.
- Global [RAIL_CalibrateIr](#) (RAIL_Handle_t railHandle, uint32_t *imageRejection)
Please use [RAIL_CalibrateIrAlt](#) instead.
- Global [RAIL_DirectModeConfig_t::dclkLoc](#)
No longer used (ignored).
- Global [RAIL_DirectModeConfig_t::dinLoc](#)
No longer used (ignored).
- Global [RAIL_DirectModeConfig_t::doutLoc](#)
No longer used (ignored).
- Global [RAIL_EVENT_TX_BLOCKED_TOO_HOT](#)
but reserved for possible future use.
- Global [RAIL_IEEE802154_CalibrateIrSubGhz](#) (RAIL_Handle_t railHandle, uint32_t *imageRejection)
Please use [RAIL_CalibrateIrAlt](#) instead.
- Global [RAIL_InitTxPowerCurves](#) (const [RAIL_TxPowerCurvesConfig_t](#) *config)
function is no longer supported. Must use [RAIL_InitTxPowerCurvesAlt\(\)](#) instead.
- Global [RAIL_PtiConfig_t::dclkLoc](#)
No longer used (ignored).
- Global [RAIL_PtiConfig_t::dframeLoc](#)
No longer used (ignored).
- Global [RAIL_PtiConfig_t::doutLoc](#)
No longer used (ignored).
- Global [RAIL_RX_CHANNEL_HOPPING_OPTION_DEFAULT](#)
Please use [RAIL_RX_CHANNEL_HOPPING_OPTIONS_DEFAULT](#) instead.
- Global [RAIL_RxChannelHoppingConfigEntry_t::delayMode](#)
Set delayMode to [RAIL_RX_CHANNEL_HOPPING_DELAY_MODE_STATIC](#).
- Global [RAIL_RxChannelHoppingDelayMode_t](#)
Set only to [RAIL_RX_CHANNEL_DELAY_MODE_STATIC](#).
- Global [RAIL_RxDutyCycleConfig_t::delayMode](#)

Set delayMode to [RAIL_RX_CHANNEL_HOPPING_DELAY_MODE_STATIC](#).

- Global [RAIL_TX_POWER_MODE_2P4_HIGHEST](#)
Please use [RAIL_TX_POWER_MODE_2P4GIG_HIGHEST](#) instead.
- Global [RAIL_TX_POWER_MODE_2P4_HP](#)
Please use [RAIL_TX_POWER_MODE_2P4GIG_HP](#) instead.
- Global [RAIL_TX_POWER_MODE_2P4_LP](#)
Please use [RAIL_TX_POWER_MODE_2P4GIG_LP](#) instead.
- Global [RAIL_TX_POWER_MODE_2P4_MP](#)
Please use [RAIL_TX_POWER_MODE_2P4GIG_MP](#) instead.
- Global [RAIL_TX_POWER_MODE_OFDM_PA](#)
Please use [RAIL_TX_POWER_MODE_OFDM_PA_POWERSETTING_TABLE](#) instead.
- Global [RAIL_TX_POWER_MODE_SUBGIG](#)
Please use [RAIL_TX_POWER_MODE_SUBGIG_HP](#) instead.
- Global [RAIL_ZWAVE_REGIONID_EU_LR_END_DEVICE](#)
Backwards compatible name.
- Global [RAIL_ZWAVE_REGIONID_US_LR_END_DEVICE](#)
Backwards compatible name.

About RAIL Examples

RAIL Examples

A number of examples are provided with Simplicity Studio and the Proprietary RAIL SDK. Each example has an associated README that explains the purpose of the example and how to use it. Some of the examples have more extensive documentation, and that is included in this section. Because new examples are always being added, the SiSDK may contain other examples not documented here.

Many of these examples are intended only to test specific aspects of the radio. They are not intended as a starting point for your own application, although they may contain code for some features you want to use. **RAIL - SoC Empty** is a recommended starting point.

RAIL - SoC Empty: The Empty Example application initializes RAIL to support a Single PHY radio configuration and defines an empty radio event handler. These are the necessary parts for every RAIL-based application.

RAIL - SoC Burst Duty Cycle: The Burst Duty Cycle example is a complete duty cycle solution implemented on top of RAIL. It implements both the Master (sender) and the Slave (listener) nodes. In receive mode the device spends most of the time in sleep state (EM1 or EM2) and wakes up periodically for short periods. In transmit mode the device transmits multiple packets in quick succession. This application can be used to optimize radio configuration and designing LDC (Low Duty-cycle) periods on the receiver in order to save as much energy as possible without PER (Packet Error Rate) reduction.

RAIL - SoC Long Preamble Duty Cycle: The Long Preamble Duty Cycle example is a complete duty cycle solution implemented on top of RAIL. This example supports both receive and transmit modes. In receive mode the device spends most of the time in a sleep state (EM1 or EM2) and wakes up periodically for a short periods. In transmit mode the device transmits one packet using long preamble. The low duty cycle mode of the EFR32 radios is a simple and effective way to reduce the current consumption of the system by autonomously waking up the receiver to look for transmitted packets, while the MCU remains in its lowest power mode.

RAIL - SoC Energy Mode: The Energy Mode example application serves as a complete, ready-to-use program with which you can accelerate the evaluation of the of energy consumption on your custom radio design by iterating over the available energy modes or using Tx/Rx radio operations.

RAIL - SoC Simple TRX: The Simple TRX example demonstrates the simplest exchange of transmit and receive operations between two nodes implemented in RAIL. It can be used to set up a simple link test, where the nodes are listening except for the short periods where packet transmission takes place.

RAIL - SoC Simple TRX with Auto-ACK: The Simple TRX with Auto-ACK example demonstrates how RAIL's Auto-ACK feature works. It can be used to set up a simple link test that requires Auto-ACK functionality.

RAIL - SoC Simple TRX Standards: The Simple TRX Standards example demonstrates the simplest packet exchange of transmit and receive operations between two nodes implemented in RAIL compatible with Bluetooth LE, IEEE 802.15.4 (2.4 GHz only) and IEEE 802.15.4g (subGHz only) standards. It can be used to set up a simple link test, where the nodes are listening except for the short periods where packet transmission takes place.

RAIL - SoC RAILtest: The RAILtest application (RAILtest) is a simple tool for testing the radio and the functionality of the RAIL library. It is fully documented in the [RAILtest User's Guide \(UG409\)](#).

RAIL - SoC Range Test: Range Test is a standalone test application that creates a radio link between devices and sends a predefined number of packets from the transmitter side to the receiver. The Range Test demo implements packet error rate (PER) measurement. PER is a commonly-used technique for measuring the quality of RF links in wireless systems under particular conditions. Range Test applications are documented in [Flex SDK v3.x Range Test Demo User's Guide \(UG471\)](#).

RAIL - SoC Wireless M-bus Meter and Collector: Wireless M-bus applications are documented in [Using RAIL for Wireless M-Bus Applications with EFR32](#).

Further example applications are available in the [SiliconLabs/proprietary_rail](#) GitHub repository. This repository includes several example applications that illustrate the use of the RAIL library in various specific scenarios. These examples are

compatible with Series 2 devices exclusively.

RAIL Proprietary - Tutorial Transmit: Application demonstrating the [Transmitting a Packet](#) tutorial.

RAIL Proprietary - Tutorial Event Handling: Application demonstrating the [Event Handling and Automatic State Transitions](#) tutorial.

RAIL Proprietary - Tutorial Downloading Messages: Application demonstrating the [Downloading Received Messages](#) tutorial.

RAIL Proprietary - Address Filtering: Application designed to demonstrate the [RAIL Address Filtering APIs](#).

RAIL Proprietary - Improved RF Sense Transmitter: Application illustrating how to enhance the Selective RF Sense range by transmitting across multiple frequencies.

RAIL Proprietary - Time Synchronization: Application demonstrating how to synchronize local time within RAIL's timebase across multiple devices.

RAIL Proprietary - Direct Mode Packet Detector: Application demonstrating how to utilize [Direct-to-PRS](#) mode for detecting packets in software.

RAIL Proprietary - Direct to Buffer: Application demonstrating how to use [Direct-to-Buffer](#) mode.

RAIL SDK Services

Services

Introduction

The RAIL SDK components offer a practical framework for building wireless communication applications. These components help developers create applications that can adapt to various communication needs and manage power consumption effectively. Whether you're working on a simple project or a complex multi-protocol system, the RAIL SDK provides the necessary tools and features to support your development.

Extension

Extension

Extension

Extensions for the RAIL SDK are components that enhance existing sample applications or custom sample applications with new features without modifying the existing code. One such extension is the [Simple OS](#) extension.

Modules

[Simple OS](#)

Simple OS

Simple OS

The [Simple OS](#) component in the RAIL SDK provides a basic structure for RAIL SDK users to update a baremetal application to an OS-based application. It supports FreeRTOS and Micrium OS, offering a seamless transition for developers who need to integrate OS functionalities into their applications.

Key Features

- **Task Initialization:** The [app_task_init\(\)](#) function initializes the proprietary application task without using any dynamic memory allocation. This ensures that the task is created with a predefined stack and task buffer, making it suitable for memory-constrained environments.
- **Task Notification:** The [app_task_notify](#) function is used to notify the kernel to allow the proprietary task to run. This function ensures that the task is only notified if it has been successfully created.
- **Static Task Allocation:** The component uses static task allocation to avoid dynamic memory allocation, which is crucial for systems with limited memory resources. The task stack and buffer are statically allocated, ensuring predictable memory usage.
- **Max Blocking Time:** The component defines a maximum blocking time for the proprietary application task, ensuring that the task does not block indefinitely.

Functions

void	app_task_init (void)	Initialize the task that will run the proprietary task than includes, the proprietary routines including app_init and app_process functions.
void	app_task_notify (void)	Notify the kernel to allow the proprietary task to run, as it is designed to wait for notifications or flags.
void	print_sample_app_name (const char *app_name)	Print the sample app name with the OS which it is running on.

Function Documentation

app_task_init

```
void app_task_init (void )
```

Initialize the task that will run the proprietary task than includes, the proprietary routines including app_init and app_process functions.

Parameters

N/A		

Definition at line 60 of file /mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_simple_os/app_task_init.h

app_task_notify

```
void app_task_notify (void )
```

Notify the kernel to allow the proprietary task to run, as it is designed to wait for notifications or flags.

Parameters

N/A		

Definition at line 66 of file /mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_simple_os/app_task_init.h

print_sample_app_name

```
void print_sample_app_name (const char * app_name)
```

Print the sample app name with the OS which it is running on.

Parameters

[in]	app_name	Sample app name to be printed.

Definition at line 73 of file /mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_simple_os/app_task_init.h

Utility

Utility

Utility

The Utility components in the RAIL SDK provide a toolbox of essential tools and functions that enhance the development and management of wireless communication applications. These utilities leverage various RAIL library features to offer additional capabilities, such as channel and PHY (Physical Layer) selection, power management integration, and packet handling. Below is a detailed description of the key components and their functionalities:

Modules

[Channel Selector](#)

[PHY Selector](#)

[Packet Assistant](#)

[Power Manager Integration](#)

Channel Selector

Channel Selector

The RAIL Channel Selector component allows the application to dynamically select the communication channel at both compile-time and runtime. This flexibility is crucial for applications that need to operate on different channels based on environmental conditions or regulatory requirements. The component provides functions to set and get the selected channel, ensuring that the application can easily switch channels as needed.

- Key functions include:
- [set_selected_channel\(\)](#): Sets the communication channel.
 - [get_selected_channel\(\)](#): Retrieves the current communication channel.
 - [restart_rx_channel\(\)](#): Restarts the RX operation on the selected channel.

Modules

[Configurations](#)

Functions

void	set_configured_channel(void) Sets the configured channel using the DEFAULT_CHANNEL define.
uint8_t	set_selected_channel(uint16_t new_channel) Sets which channel the app will communicate on.
uint16_t	get_selected_channel(void) Get which channel the app will communicate on.
sl_rail_status_t	restart_rx_channel(void) Restarts the RX operation on the selected channel.

Function Documentation

set_configured_channel

```
void set_configured_channel (void )
```

Sets the configured channel using the [DEFAULT_CHANNEL](#) define.

Parameters

N/A		

Sets the configured channel to [DEFAULT_CHANNEL](#)

Definition at line 71 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_channel_selector/sl_rail_sdk_channel_selector.h

set_selected_channel

```
uint8_t set_selected_channel (uint16_t new_channel)
```

Sets which channel the app will communicate on.

Parameters

[in]	new_channel	requested new channel number

Tests the channel number with configured channel limits from radio configurator, if it is invalid first valid channel will be used. Also restarts the receiving by calling [restart_rx_channel\(\)](#)

Returns

- 1 if channel number is valid else 0

Definition at line 83 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_channel_selector/sl_rail_sdk_channel_selector.h

get_selected_channel

```
uint16_t get_selected_channel (void )
```

Get which channel the app will communicate on.

Parameters

N/A		

Returns

- current channel number used for communication

Definition at line 90 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_channel_selector/sl_rail_sdk_channel_selector.h

restart_rx_channel

```
sl_rail_status_t restart_rx_channel (void )
```

Restarts the RX operation on the selected channel.

Parameters

N/A		

Returns

- status of the sl_rail_start_rx() function

Definition at line 97 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_channel_selector/sl_rail_sdk_channel_selector.h

Configurations

Configurations

Macros

```
#define    DEFAULT_CHANNEL 0
Default channel number to start the app.
```

PHY Selector

PHY Selector

The RAIL PHY Selector component allows the application to select the PHY configuration for communication. This component is useful for applications that need to switch between different PHY settings to optimize performance or comply with different communication standards. The component provides functions to set and get the selected PHY, ensuring that the application can easily switch PHY configurations as needed.

- Key functions include:
- `set_selected_phy()`: Sets the PHY configuration for communication.
 - `get_selected_phy()`: Retrieves the current PHY configuration.

Functions

```
uint8_t    set_selected_phy(uint16_t new_phy)
           Set the PHY the app will use.

uint16_t    get_selected_phy(void)
           Get the currently selected PHY number.
```

Function Documentation

set_selected_phy

```
uint8_t set_selected_phy (uint16_t new_phy)
```

Set the PHY the app will use.

Parameters

[in]	new_phy	The requested new PHY number.

Returns

- 1 if the PHY number is valid, otherwise 0.

Definition at line 71 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_phy_selector/sl_rail_sdk_phy_selector.h

get_selected_phy

```
uint16_t get_selected_phy (void )
```

Get the currently selected PHY number.

Parameters

N/A		

Returns

- The selected PHY number.

Definition at line 77 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_phy_selector/sl_rail_sdk_phy_selector.h

Packet Assistant

Packet Assistant

The RAIL Packet Assistant component provides functions to prepare and unpack packets for various protocols such as Wi-SUN FSK, Wi-SUN OFDM, SUN OQPSK, and Sidewalk. This component ensures that the application can generate the appropriate packet headers for any selected PHY, allowing RAIL to send out the proper packet with the user payload. The component provides functions to prepare and unpack packets, ensuring that the application can handle different packet formats as needed.

Key functions include:

- [prepare_packet\(\)](#): Prepares the packet for sending and loads it into the RAIL TX FIFO.
- [unpack_packet\(\)](#): Unpacks the received packet, points to the payload, and returns the length.

Modules

[Configurations](#)

[Type definitions](#)

Functions

void	validation_check (void) Checks PHY setting to avoid errors at packet sending.
void	update_assistant_pointers (uint8_t new_phy_index) Updates assistant pointers with a new PHY index.
uint16_t	unpack_packet (sl_rail_handle_t rail_handle, uint8_t *rx_destination, const sl_rail_rx_packet_info_t *packet_information, uint8_t **start_of_payload) Unpacks the received packet, points to the payload, and returns the length.
void	prepare_packet (sl_rail_handle_t rail_handle, uint8_t *out_data, uint16_t length) Prepares the packet for sending and load it in the RAIL TX FIFO.
uint8_t	get_print_packet_info (void) Get the print packet information.
void	set_print_packet_info (uint8_t new_print_packet_info) Set the print packet information.
void	printf_rx_packet (const uint8_t *const rx_buffer, uint16_t length) Prints out the received packet.
uint8_t	get_wisun_ofdm_rate (void) Get the Wi-SUN OFDM rate.
uint8_t	set_wisun_ofdm_rate (uint8_t new_rate) Set the Wi-SUN OFDM rate.
uint8_t	get_wisun_ofdm_scrambler (void) Get the Wi-SUN OFDM scrambler.
uint8_t	set_wisun_ofdm_scrambler (uint8_t new_scrambler) Set the Wi-SUN OFDM scrambler.

uint8_t	get_wisun_fsk_fcs (void) Get the Wi-SUN FSK FCS value.
uint8_t	set_wisun_fsk_fcs (uint8_t new_fcs) Set the Wi-SUN FSK FCS value.
uint8_t	get_wisun_fsk_whitening (void) Get the Wi-SUN FSK Whitening value.
uint8_t	set_wisun_fsk_whitening (uint8_t new_whitening) Set the Wi-SUN FSK Whitening value.
uint8_t	get_sun_oqpsk_spreading_mode (void) Get the Sun OQPSK Spreading Mode.
uint8_t	set_sun_oqpsk_spreading_mode (uint8_t new_spreading_mode) Set the Sun OQPSK Spreading Mode.
uint8_t	get_sun_oqpsk_rate_mode (void) Get the Sun OQPSK Rate Mode.
uint8_t	set_sun_oqpsk_rate_mode (uint8_t new_rate_mode) Set the Sun OQPSK Rate Mode.
uint8_t	get_sidewalk_fsk_type (void) Get the Sidewalk FSK FCS value.
uint8_t	set_sidewalk_fsk_type (uint8_t new_fcs) Set the Sidewalk FSK FCS value.
uint8_t	get_sidewalk_whitening (void) Get the Sidewalk Whitening value.
uint8_t	set_sidewalk_whitening (uint8_t new_whitening) Set the Sidewalk Whitening value.

Function Documentation

validation_check

```
void validation_check (void )
```

Checks PHY setting to avoid errors at packet sending.

Parameters

N/A		

Definition at line 106 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

update_assistant_pointers

```
void update_assistant_pointers (uint8_t new_phy_index)
```

Updates assistant pointers with a new PHY index.

Parameters

[in]	new_phy_index	The new PHY index to update.

Definition at line 112 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

unpack_packet

```
uint16_t unpack_packet (sl_rail_handle_t rail_handle, uint8_t * rx_destination, const sl_rail_rx_packet_info_t * packet_information, uint8_t ** start_of_payload)
```

Unpacks the received packet, points to the payload, and returns the length.

Parameters

[in]	rail_handle	The RAIL handle used for unpacking the packet.
[out]	rx_destination	Where should the full packet be unpacked.
[out]	packet_information	Where should all the information of the packet stored.
[out]	start_of_payload	Pointer where the payload starts.

Returns

- The length of the received payload.

Definition at line 123 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

prepare_packet

```
void prepare_packet (sl_rail_handle_t rail_handle, uint8_t * out_data, uint16_t length)
```

Prepares the packet for sending and load it in the RAIL TX FIFO.

Parameters

[in]	rail_handle	Which rail handlers should be used for the TX FIFO writing
[in]	out_data	The payload buffer
[in]	length	The length of the payload

Definition at line 132 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

get_print_packet_info

```
uint8_t get_print_packet_info (void )
```

Get the print packet information.

Parameters

N/A		

This function retrieves the current print packet information setting. **Returns**

- The current print packet information setting.

Definition at line 139 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

set_print_packet_info

```
void set_print_packet_info (uint8_t new_print_packet_info)
```

Set the print packet information.

Parameters

[in]	new_print_packet_info	The new print packet information setting to be applied.

This function sets a new value for the print packet information setting.

Definition at line 146 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

printf_rx_packet

```
void printf_rx_packet (const uint8_t *const rx_buffer, uint16_t length)
```

Prints out the received packet.

Parameters

[in]	rx_buffer	The buffer where the packet is stored.
[in]	length	The length of the packet.

This function prints the contents of the received packet.

Definition at line 154 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

get_wisun_ofdm_rate

```
uint8_t get_wisun_ofdm_rate (void )
```

Get the Wi-SUN OFDM rate.

Parameters

N/A		

Returns the current Wi-SUN OFDM rate. **Returns**

- The Wi-SUN OFDM rate.

Definition at line 161 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

set_wisun_ofdm_rate

```
uint8_t set_wisun_ofdm_rate (uint8_t new_rate)
```

Set the Wi-SUN OFDM rate.

Parameters

[in]	new_rate	The new Wi-SUN OFDM rate to set.

Sets the Wi-SUN OFDM rate to the specified value. **Returns**

- 1 if the value was set successfully, 0 otherwise.

Definition at line 169 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

get_wisun_ofdm_scrambler

uint8_t get_wisun_ofdm_scrambler (void)

Get the Wi-SUN OFDM scrambler.

Parameters

N/A		

Returns the current Wi-SUN OFDM scrambler. **Returns**

- The Wi-SUN OFDM scrambler.

Definition at line 176 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

set_wisun_ofdm_scrambler

uint8_t set_wisun_ofdm_scrambler (uint8_t new_scrambler)

Set the Wi-SUN OFDM scrambler.

Parameters

[in]	new_scrambler	The new Wi-SUN OFDM scrambler to set, 2 bits wide (0-3)

Sets the Wi-SUN OFDM scrambler to the specified value. **Returns**

- 1 if the value was set successfully, 0 otherwise.

Definition at line 184 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

get_wisun_fsk_fcs

uint8_t get_wisun_fsk_fcs (void)

Get the Wi-SUN FSK FCS value.

Parameters

N/A		

Returns the current Wi-SUN FSK FCS value. **Returns**

- The Wi-SUN FSK FCS value.

Definition at line 191 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

set_wisun_fsk_fcs

```
uint8_t set_wisun_fsk_fcs (uint8_t new_fcs)
```

Set the Wi-SUN FSK FCS value.

Parameters

[in]	new_fcs	The new Wi-SUN FSK FCS value to set.

Sets the Wi-SUN FSK FCS value to the specified new value. **Returns**

- 1 if the value was set successfully, 0 otherwise.

Definition at line 199 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

get_wisun_fsk_whitening

```
uint8_t get_wisun_fsk_whitening (void )
```

Get the Wi-SUN FSK Whitening value.

Parameters

N/A		

Returns the current Wi-SUN FSK Whitening value. **Returns**

- The Wi-SUN FSK Whitening value.

Definition at line 206 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

set_wisun_fsk_whitening

```
uint8_t set_wisun_fsk_whitening (uint8_t new_whitening)
```

Set the Wi-SUN FSK Whitening value.

Parameters

[in]	new_whitening	The new Wi-SUN FSK Whitening value to set.

Sets the Wi-SUN FSK Whitening value to the specified new value. **Returns**

- 1 if the value was set successfully, 0 otherwise.

Definition at line 214 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

get_sun_oqpsk_spreading_mode

```
uint8_t get_sun_oqpsk_spreading_mode (void )
```

Get the Sun OQPSK Spreading Mode.

Parameters

N/A		

Retrieves the current Sun OQPSK Spreading Mode. **Returns**

- The current Sun OQPSK Spreading Mode.

Definition at line 221 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

set_sun_oqpsk_spreading_mode

```
uint8_t set_sun_oqpsk_spreading_mode (uint8_t new_spreading_mode)
```

Set the Sun OQPSK Spreading Mode.

Parameters

[in]	new_spreading_mode	The new Sun OQPSK Spreading Mode to set.

Sets the Sun OQPSK Spreading Mode to the specified new mode. **Returns**

- 1 if the value was set successfully, 0 otherwise.

Definition at line 229 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

get_sun_oqpsk_rate_mode

```
uint8_t get_sun_oqpsk_rate_mode (void )
```

Get the Sun OQPSK Rate Mode.

Parameters

N/A		

Retrieves the current Sun OQPSK Rate Mode. **Returns**

- The current Sun OQPSK Rate Mode.

Definition at line 236 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

set_sun_oqpsk_rate_mode

```
uint8_t set_sun_oqpsk_rate_mode (uint8_t new_rate_mode)
```

Set the Sun OQPSK Rate Mode.

Parameters

[in]	new_rate_mode	The new Sun OQPSK Rate Mode to set, 2 bits wide (0-3)

Sets the Sun OQPSK Rate Mode to the specified new mode. **Returns**

- 1 if the value was set successfully, 0 otherwise.

Definition at line 244 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

get_sidewalk_fcs_type

uint8_t get_sidewalk_fcs_type (void)

Get the Sidewalk FSK FCS value.

Parameters

N/A		

Retrieves the current Sidewalk FSK FCS value. **Returns**

- The current Sidewalk FSK FCS value.

Definition at line 251 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

set_sidewalk_fcs_type

uint8_t set_sidewalk_fcs_type (uint8_t new_fcs)

Set the Sidewalk FSK FCS value.

Parameters

[in]	new_fcs	The new Sidewalk FSK FCS value to set.

Sets the Sidewalk FSK FCS value to the specified new value. **Returns**

- 1 if the value was set successfully, 0 otherwise.

Definition at line 259 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

get_sidewalk_whitening

uint8_t get_sidewalk_whitening (void)

Get the Sidewalk Whitening value.

Parameters

N/A		

Retrieves the current Sidewalk Whitening value. **Returns**

- The current Sidewalk Whitening value.

Definition at line 266 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

set_sidewalk_whitening

```
uint8_t set_sidewalk_whitening (uint8_t new_whitening)
```

Set the Sidewalk Whitening value.

Parameters

[in]	new_whitening	The new Sidewalk Whitening value to set.

Sets the Sidewalk Whitening value to the specified new value. **Returns**

- 1 if the value was set successfully, 0 otherwise.

Definition at line 274 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

Configurations

Configurations

Macros

#define	PRINT_PACKET_INFO (0) Enable assistant log prints.
#define	WISUN_FSK_FCS_TYPE (0) Wi-SUN FCS is on/off.
#define	WISUN_FSK_WHITENING (1) Wi-SUN Whitening is on/off.
#define	WISUN_OFDM_RATE (6) Wi-SUN OFDM rate Configuration.
#define	WISUN_OFDM_SCRAMBLER (0) Wi-SUN OFDM Scrambler Configuration.
#define	SUN_OQPSK_SPREADINGMODE (0) SUN OQPSK spreading mode Configuration.
#define	SUN_OQPSK_RATEMODE (0) SUN OQPSK rate mode Configuration.
#define	SIDEWALK_FSK_FCS_TYPE (1) SideWalk FSK FCS is on/off.
#define	SIDEWALK_FSK_WHITENING (1) SideWalk FSK Whitening is on/off.

Type definitions

Type definitions

Enumerations

```
enum RAIL_SDK_Protocol_t {
    CUSTOM_AND_SUN_OQPSK = 0
    EMBER_PHY = 1
    THREAD = 2
    BLE = 3
    CONNECT = 4
    ZIGBEE = 5
    ZWAVE = 6
    WISUN = 7
    BTC = 8
    SIDEWALK = 9
    LONGRANGE = 10
    MBUS = 11
    SIGFOX = 12
    UNDEFINED = 13
}
```

Enumeration of different radio protocols.

Enumeration Documentation

RAIL_SDK_Protocol_t

RAIL_SDK_Protocol_t

Enumeration of different radio protocols.

This enumeration defines the various radio protocols that can be used.

Enumerator	
CUSTOM_AND_SUN_OQPSK	Custom and SUN OQPSK protocol
EMBER_PHY	Ember PHY protocol
THREAD	Thread protocol
BLE	Bluetooth Low Energy protocol
CONNECT	Connect protocol, original, SUN-FSK and OFDM
ZIGBEE	Zigbee protocol
ZWAVE	Z-Wave protocol
WISUN	Wi-SUN protocol, FAN 1.0 and FAN 1.1 with OFDM
BTC	BTC protocol
SIDEWALK	Sidewalk protocol
LONGRANGE	Long-range protocol
MBUS	Wireless M-Bus protocol
SIGFOX	Sigfox protocol
UNDEFINED	Undefined protocol

Definition at line 78 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_packet_assistant/sl_rail_sdk_packet_assistant.h

Power Manager Integration

Power Manager Integration

The Power Manager Integration component integrates RAIL with the Power Manager to enable efficient power management. This integration ensures that the RAIL and Power Manager work together in a synchronized mode, allowing the application to manage power consumption effectively. The component provides functions to initialize the power management integration, ensuring that the application can enter and exit low-power states as needed.

Key functions include:

- `sl_rail_sdk_sleep_init()`: Prepares the RAIL and Power Manager to work together in a synchronized mode.

Functions

`sl_rail_status_t` `sl_rail_sdk_sleep_init(void)`
Initializes RAIL and Power Manager to work together in SYNC mode.

Function Documentation

sl_rail_sdk_sleep_init

`sl_rail_status_t sl_rail_sdk_sleep_init (void )`

Initializes RAIL and Power Manager to work together in SYNC mode.

Parameters

N/A		

This function sets up the necessary configurations and prepares the RAIL SDK sleep module for operation. It should be called before using any other functions in the sleep module.

Returns

- `sl_rail_status_t`
 - `SL_RAIL_STATUS_NO_ERROR` if initialization was successful.
 - An appropriate error code otherwise.

Definition at line 68 of file /mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_sleep/sl_rail_sdk_sleep.h

Wireless M-Bus

Wireless M-Bus

Wireless M-Bus Support

The Wireless M-Bus components in the RAIL SDK provide a comprehensive suite of tools and functionalities for developing and managing wireless communication applications that adhere to the Wireless M-Bus standard. These components are designed to facilitate the implementation of Wireless M-Bus communication in various types of devices, such as meters and sensors, by offering support for packet handling, encryption, channel and PHY selection, and power management. Below is a detailed description of the key components and their functionalities.

Modules

[Wireless M-Bus Button Pulse Counter](#)

[Wireless M-Bus Sensor Core](#)

[Wireless M-Bus Support](#)

[Wireless M-Bus Thermometer Sensor](#)

[Wireless M-Bus Virtual Water Meter Sensor](#)

Wireless M-Bus Button Pulse Counter

Wireless M-Bus Button Pulse Counter

The Wireless M-Bus Pulse Counter component is part of the RAIL SDK's Wireless M-Bus suite, designed to count and send button presses between transmission frames. This component is specifically tailored for Wireless M-Bus meters, providing a solution for integrating pulse counter data into the Wireless M-Bus communication framework.

Key Features:

- **Pulse Counting:** Making button as a virtual pulse counter.

Modules

[Configurations](#)

Functions

sl_status_t	sl_rail_sdk_wmbus_sensor_pulse_counter_init (void) Wireless M-Bus Pulse counter initialization function.
sl_status_t	sl_rail_sdk_wmbus_sensor_pulse_counter_deinit (void) Wireless M-Bus Pulse counter deinitialization function.
sl_status_t	sl_rail_sdk_wmbus_sensor_pulse_counter_measure (void) Wireless M-Bus Pulse counter measure function.
sl_rail_sdk_wmbus_sensor_data_t *	sl_rail_sdk_wmbus_sensor_pulse_counter_get_data (void) Retrieves the sensor data for the Wireless M-Bus pulse counter sensor.
sl_status_t	sl_rail_sdk_wmbus_sensor_pulse_counter_print (void) Wireless M-Bus Pulse counter 7 segment LCD print function.
sl_status_t	sl_rail_sdk_wmbus_sensor_pulse_counter_button_pressed (void) Wireless M-Bus Pulse counter button pressed function.

Function Documentation

sl_rail_sdk_wmbus_sensor_pulse_counter_init

sl_status_t sl_rail_sdk_wmbus_sensor_pulse_counter_init (void)

Wireless M-Bus Pulse counter initialization function.

Parameters

N/A		

Returns

- sl_status_t: SL_STATUS_OK if init was successful SL_STATUS_FAIL if init failed

Definition at line 67 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_pulse_counter/sl_rail_sdk_wmbus_sensor_pulse_counter.h

sl_rail_sdk_wmbus_sensor_pulse_counter_deinit

```
sl_status_t sl_rail_sdk_wmbus_sensor_pulse_counter_deinit (void )
```

Wireless M-Bus Pulse counter deinitialization function.

Parameters

N/A		

Returns

- sl_status_t: SL_STATUS_OK if init was successful SL_STATUS_FAIL if init failed

Definition at line 75 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_pulse_counter/sl_rail_sdk_wmbus_sensor_pulse_counter.h

sl_rail_sdk_wmbus_sensor_pulse_counter_measure

```
sl_status_t sl_rail_sdk_wmbus_sensor_pulse_counter_measure (void )
```

Wireless M-Bus Pulse counter measure function.

Parameters

N/A		

Returns

- sl_status_t: SL_STATUS_OK if measure was successful SL_STATUS_FAIL if measure failed

Definition at line 83 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_pulse_counter/sl_rail_sdk_wmbus_sensor_pulse_counter.h

sl_rail_sdk_wmbus_sensor_pulse_counter_get_data

```
sl_rail_sdk_wmbus_sensor_data_t * sl_rail_sdk_wmbus_sensor_pulse_counter_get_data (void )
```

Retrieves the sensor data for the Wireless M-Bus pulse counter sensor.

Parameters

N/A		

Returns a pointer to the [sl_rail_sdk_wmbus_sensor_data_t](#) structure that contains the sensor data for the Wireless M-Bus pulse counter sensor.

Returns

- Pointer to a [sl_rail_sdk_wmbus_sensor_data_t](#), the sensor data for the Wireless M-Bus pulse counter sensor

Definition at line 94 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_pulse_counter/sl_rail_sdk_wmbus_sensor_pulse_counter.h

sl_rail_sdk_wmbus_sensor_pulse_counter_print

```
sl_status_t sl_rail_sdk_wmbus_sensor_pulse_counter_print (void )
```

Wireless M-Bus Pulse counter 7 segment LCD print function.

Parameters

N/A		

Returns

- sl_status_t: SL_STATUS_OK if 7 segment LCD print was successfully SL_STATUS_FAIL if 7 segment LCD print failed

Definition at line 102 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_pulse_counter/sl_rail_sdk_wmbus_sensor_pulse_counter.h

sl_rail_sdk_wmbus_sensor_pulse_counter_button_pressed

```
sl_status_t sl_rail_sdk_wmbus_sensor_pulse_counter_button_pressed (void )
```

Wireless M-Bus Pulse counter button pressed function.

Parameters

N/A		

This function handles the button press event for the Wireless M-Bus pulse counter, in this case, it increments the counter value.

Returns

- sl_status_t: SL_STATUS_OK if button press was successfully SL_STATUS_FAIL if button press failed

Definition at line 112 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_pulse_counter/sl_rail_sdk_wmbus_sensor_pulse_counter.h

Configurations

Configurations

Macros

```
#define SL_RAIL_SDK_WMBUS_SENSOR_PULSE_COUNTER_SLEEPTIMER_PRIORITY 10
Priority of the sleeptimer used to periodically read the pulse counter.

#define SL_RAIL_SDK_WMBUS_SENSOR_PULSE_COUNTER_SLEEPTIMER_TIMEOUT 1000
Timeout of the sleeptimer used to periodically read the pulse counter.
```

Wireless M-Bus Sensor Core

Wireless M-Bus Sensor Core

The Wireless M-Bus Sensor Core component is designed for use with Wireless M-Bus meters, providing the necessary tools to integrate various sensors into the Wireless M-Bus communication framework. This component simplifies the process of adding sensor data to Wireless M-Bus packets and ensures that the data is correctly formatted and transmitted.

- Key Features:**
- **Sensor Initialization:** Functions to initialize Wireless M-Bus sensors, ensuring they are correctly configured for communication.
 - **Data Handling:** Functions to retrieve and format sensor data for inclusion in Wireless M-Bus packets.
 - **Packet Assembly:** Tools to assemble packets with sensor data, ensuring that the data is correctly formatted and transmitted.

Modules

- [Configurations](#)
- [Type definitions](#)

Functions

sl_rail_sdk_wmbus_sensor_t *	sl_rail_sdk_wmbus_sensor_core_get_active_sensor(void) Get the active sensor.
sl_status_t	sl_rail_sdk_wmbus_sensor_core_set_active_sensor(uint8_t id) Set the active sensor.
sl_rail_sdk_wmbus_sensor_t *	sl_rail_sdk_wmbus_sensor_core_list_available_sensors(void) Get the list of available sensors.
size_t	sl_rail_sdk_wmbus_sensor_core_get_sensor_count(void) Get the number of available sensors.
sl_status_t	sl_rail_sdk_wmbus_sensor_core_init(void) Initialize the Wireless M-Bus sensor core.
void	sl_rail_sdk_wmbus_sensor_core_lcd_init(void) Initialize the Wireless M-Bus sensor core LCD.
uint16_t	sl_rail_sdk_wmbus_sensor_core_process(uint8_t *tx_buffer, uint8_t *access_number) Process the Wireless M-Bus sensor core.

Function Documentation

sl_rail_sdk_wmbus_sensor_core_get_active_sensor

```
sl_rail_sdk_wmbus_sensor_t * sl_rail_sdk_wmbus_sensor_core_get_active_sensor (void )
```

Get the active sensor.

Parameters

N/A		

Returns

- [sl_rail_sdk_wmbus_sensor_t*](#) Pointer to the active sensor.

Definition at line 159 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

sl_rail_sdk_wmbus_sensor_core_set_active_sensor

sl_status_t sl_rail_sdk_wmbus_sensor_core_set_active_sensor (uint8_t id)

Set the active sensor.

Parameters

[in]	id	Sensor ID

Returns

- sl_status_t: SL_STATUS_OK if successful, SL_STATUS_INVALID_PARAMETER if the sensor ID is invalid

Definition at line 169 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

sl_rail_sdk_wmbus_sensor_core_list_available_sensors

sl_rail_sdk_wmbus_sensor_t * sl_rail_sdk_wmbus_sensor_core_list_available_sensors (void)

Get the list of available sensors.

Parameters

N/A		

Returns

- [sl_rail_sdk_wmbus_sensor_t*](#): Pointer to the list of available sensors

Definition at line 176 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

sl_rail_sdk_wmbus_sensor_core_get_sensor_count

size_t sl_rail_sdk_wmbus_sensor_core_get_sensor_count (void)

Get the number of available sensors.

Parameters

N/A		

Returns

- size_t: Number of available sensors

Definition at line 183 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

sl_rail_sdk_wmbus_sensor_core_init

```
sl_status_t sl_rail_sdk_wmbus_sensor_core_init (void )
```

Initialize the Wireless M-Bus sensor core.

Parameters		
N/A		

Calls the initialization function of the active sensor.

- Returns
- sl_status_t: SL_STATUS_OK if successful, SL_STATUS_FAILED if error

Definition at line 192 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

sl_rail_sdk_wmbus_sensor_core_lcd_init

```
void sl_rail_sdk_wmbus_sensor_core_lcd_init (void )
```

Initialize the Wireless M-Bus sensor core LCD.

Parameters		
N/A		

Definition at line 197 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

sl_rail_sdk_wmbus_sensor_core_process

```
uint16_t sl_rail_sdk_wmbus_sensor_core_process (uint8_t * tx_buffer, uint8_t * access_number)
```

Process the Wireless M-Bus sensor core.

Parameters		
[in]	tx_buffer	The buffer to store the transmitted data.
[out]	access_number	The access number for the Wireless M-Bus packet.

Calls the packet assembler functions of the active sensor. It also passes the tx_buffer, access_number, and Wireless M-Bus accessibility to the packet assembler function.

- Returns
- The result of the packet assembler function.

Definition at line 209 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

Configurations

Configurations

Macros

```
#define SL_RAIL_SDK_WMBUS_SENSOR_ACTIVE_SENSOR_ID 0
Default active sensor index.
```

Type definitions

Type definitions

Modules

[sl_rail_sdk_wmbus_sensor_t](#)

Typedefs

<code>typedef sl_status_t(*</code>	<code>sl_rail_sdk_wmbus_sensor_init_t)(void)</code> Typedef for a function pointer to initialize the Wireless M-Bus sensor.
<code>typedef sl_status_t(*</code>	<code>sl_rail_sdk_wmbus_sensor_deinit_t)(void)</code> Typedef for a function pointer used to deinitialize the Wireless M-Bus sensor.
<code>typedef sl_status_t(*</code>	<code>sl_rail_sdk_wmbus_sensor_measure_t)(void)</code> Typedef for a function pointer that performs a sensor measurement.
<code>typedef sl_rail_sdk_wmbus _sensor_data_t * (*</code>	<code>sl_rail_sdk_wmbus_sensor_get_data_t)(void)</code> Function pointer type for getting Wireless M-Bus sensor data.
<code>typedef sl_status_t(*</code>	<code>sl_rail_sdk_wmbus_sensor_print_t)(void)</code> Typedef for a function pointer used to print sensor information.
<code>typedef sl_status_t(*</code>	<code>sl_rail_sdk_wmbus_sensor_button_pressed_t)(void)</code> Typedef for a function pointer that handles button press events.

Typedef Documentation

sl_rail_sdk_wmbus_sensor_init_t

```
sl_rail_sdk_wmbus_sensor_init_t )(void)
```

Typedef for a function pointer to initialize the Wireless M-Bus sensor.

Defines a function pointer type for a function that initializes the Wireless M-Bus sensor. Returns a status of type `sl_status_t`.

Returns

- `sl_status_t` Status code indicating the result of the initialization.

Definition at line 69 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

sl_rail_sdk_wmbus_sensor_deinit_t

```
sl_rail_sdk_wmbus_sensor_deinit_t )(void)
```


Typedef for a function pointer used to deinitialize the Wireless M-Bus sensor.

Defines a function pointer type for a function that deinitialize the Wireless M-Bus sensor. Returns a status of type `sl_status_t`.

Returns

- `sl_status_t` Status code indicating the result of the deinitialization.

Definition at line 80 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

sl_rail_sdk_wmbus_sensor_measure_t

```
sl_rail_sdk_wmbus_sensor_measure_t )(void)
```

Typedef for a function pointer that performs a sensor measurement.

Defines a function pointer type for a function that performs a sensor measurement and returns a status code of type `sl_status_t`.

Returns

- `sl_status_t` Status code indicating the result of the measurement.

Definition at line 91 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

sl_rail_sdk_wmbus_sensor_get_data_t

```
sl_rail_sdk_wmbus_sensor_get_data_t )(void)
```

Function pointer type for getting Wireless M-Bus sensor data.

Defines a function pointer type for a function that returns a pointer to a [sl_rail_sdk_wmbus_sensor_data_t](#) structure.

Returns

- Pointer to a [sl_rail_sdk_wmbus_sensor_data_t](#) structure.

Definition at line 102 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

sl_rail_sdk_wmbus_sensor_print_t

```
sl_rail_sdk_wmbus_sensor_print_t )(void)
```

Typedef for a function pointer used to print sensor information.

Defines a function pointer type for functions that print sensor information in the Wireless M-Bus sensor core SDK.

Returns

- A status code of type `sl_status_t` indicating the result of the print operation.

Definition at line 113 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

sl_rail_sdk_wmbus_sensor_button_pressed_t

```
sl_rail_sdk_wmbus_sensor_button_pressed_t )(void)
```

Typedef for a function pointer that handles button press events.

Defines a function pointer type for a function that is called when a button is pressed in the Wireless M-Bus sensor application.

Returns

- A status code of type sl_status_t indicating the result of the button press handling.

Definition at line 124 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

sl_rail_sdk_wmbus_sensor_t

Structure representing a Wireless M-Bus sensor.

This structure contains function pointers to initialize the sensor, deinitialize the sensor, measure data, get data, print data on a 7 segment LCD display, and handle button press events.

Public Attributes

<code>uint8_t</code>	<code>id</code> Sensor ID.
<code>const char *</code>	<code>name</code> Name of the sensor.
<code>sl_rail_sdk_wmbus_sensor_init_t</code>	<code>init_fnc_ptr</code> Function pointer to initialize the sensor.
<code>sl_rail_sdk_wmbus_sensor_init_t</code>	<code>deinit_fnc_ptr</code> Function pointer to deinitialize the sensor.
<code>sl_rail_sdk_wmbus_sensor_measure_t</code>	<code>measure_fnc_ptr</code> Function pointer to measure data.
<code>sl_rail_sdk_wmbus_sensor_get_data_t</code>	<code>get_data_fnc_ptr</code> Function pointer to get data.
<code>sl_rail_sdk_wmbus_sensor_print_t</code>	<code>print_fnc_ptr</code> Function pointer to print data on an 7 segment LCD display.
<code>sl_rail_sdk_wmbus_sensor_button_pressed_t</code>	<code>button_pressed_fnc_ptr</code> Function pointer to handle button press.

Public Attribute Documentation

id

```
uint8_t sl_rail_sdk_wmbus_sensor_t::id
```

Sensor ID.

Definition at line 134 of file `/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h`

name

```
const char* sl_rail_sdk_wmbus_sensor_t::name
```

Name of the sensor.

Definition at line 135 of file `/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h`

init_fnc_ptr

```
sl_rail_sdk_wmbus_sensor_init_t sl_rail_sdk_wmbus_sensor_t::init_fnc_ptr
```

Function pointer to initialize the sensor.

Definition at line 136 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

deinit_fnc_ptr

```
sl_rail_sdk_wmbus_sensor_init_t sl_rail_sdk_wmbus_sensor_t::deinit_fnc_ptr
```

Function pointer to deinitialize the sensor.

Definition at line 137 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

measure_fnc_ptr

```
sl_rail_sdk_wmbus_sensor_measure_t sl_rail_sdk_wmbus_sensor_t::measure_fnc_ptr
```

Function pointer to measure data.

Definition at line 138 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

get_data_fnc_ptr

```
sl_rail_sdk_wmbus_sensor_get_data_t sl_rail_sdk_wmbus_sensor_t::get_data_fnc_ptr
```

Function pointer to get data.

Definition at line 139 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

print_fnc_ptr

```
sl_rail_sdk_wmbus_sensor_print_t sl_rail_sdk_wmbus_sensor_t::print_fnc_ptr
```

Function pointer to print data on an 7 segment LCD display.

Definition at line 140 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

button_pressed_fnc_ptr

```
sl_rail_sdk_wmbus_sensor_button_pressed_t sl_rail_sdk_wmbus_sensor_t::button_pressed_fnc_ptr
```

Function pointer to handle button press.

Definition at line 141 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_core/sl_rail_sdk_wmbus_sensor_core.h

Wireless M-Bus Support

Wireless M-Bus Support

The Wireless M-Bus Support component adds minimalist support for coding and decoding Wireless M-Bus packets. This component is essential for applications that need to communicate using the Wireless M-Bus protocol, providing the necessary tools to handle packet formatting and interpretation.

Key Features:

- **Packet Coding and Decoding:** Functions to encode and decode Wireless M-Bus packets, ensuring correct data formatting for transmission and reception.
- **Function Codes:** Defines a set of function codes used in the first block's C-field, such as [WMBUS_FUNCTION_SND_NKE](#), [WMBUS_FUNCTION_SND_UD](#), and [WMBUS_FUNCTION_SND_NR](#).
- **Device Types:** Supports various device types used in address fields, including oil meters, electricity meters, gas meters, and more.
- **Manufacturer ID:** Functions to retrieve the manufacturer ID (M-field) for M-Bus addressing.
- **Encryption and Decryption:** Includes functions for encrypting and decrypting Wireless M-Bus frames using the Crypto5 algorithm.

Modules

[Configurations](#)

[Type definitions](#)

Functions

uint16_t	sl_rail_sdk_wmbus_phy_software (uint8_t *buffer, uint8_t length, uint16_t buffer_length) Runs the required software based phy processes on the buffer, depending on the mode.
sl_rail_sdk_wmbus_accessibility_t	sl_rail_sdk_wmbus_get_accessibility (void) Get the current Wireless M-Bus accessibility settings.
void	sl_rail_sdk_wmbus_set_accessibility (sl_rail_sdk_wmbus_accessibility_t new_wmbus_accessibility) Set the current Wireless M-Bus accessibility settings.
void	sl_rail_sdk_wmbus_init (void) Initializes the MBUS and its mode.
void	sl_rail_sdk_wmbus_set_mode (sl_rail_sdk_wmbus_mode_t mode) Sets the MBUS mode to use.
sl_rail_sdk_wmbus_mode_t	sl_rail_sdk_wmbus_get_selected_mode (void) Gets the selected Wireless M-Bus mode.
uint32_t	sl_rail_sdk_wmbus_get_meter_limited_acc_rx_start (bool slowMode) Returns the delay after TX when RX should start in limited accessibility sl_rail_sdk_wmbus_set_mode should be called before using this function.
uint32_t	sl_rail_sdk_wmbus_get_meter_limited_acc_rx_stop (bool slowMode) Returns the delay after TX when RX should stop in limited accessibility sl_rail_sdk_wmbus_set_mode should be called before using this function.

uint8_t	sl_rail_sdk_wmbus_frame_add_payload_int8 (uint8_t *buffer, sl_rail_sdk_wmbus_frame_dib_function_t dibFunction, uint8_t vib, int8_t value) Helper function for building the payload of a Wireless M-Bus frame.
uint8_t	sl_rail_sdk_wmbus_frame_add_payload_int16 (uint8_t *buffer, sl_rail_sdk_wmbus_frame_dib_function_t dibFunction, uint8_t vib, int16_t value) Helper function for building the payload of a Wireless M-Bus frame.
uint8_t	sl_rail_sdk_wmbus_frame_add_payload_int32 (uint8_t *buffer, sl_rail_sdk_wmbus_frame_dib_function_t dibFunction, uint8_t vib, int32_t value) Helper function for building the payload of a Wireless M-Bus frame.
uint16_t	sl_rail_sdk_wmbus_frame_add_idle_filler (uint8_t *buffer, uint16_t fillerLength) Helper function for building the payload of a Wireless M-Bus frame.
uint16_t	sl_rail_sdk_wmbus_frame_chars_to_mfield (const char *manuf) Helper function for converting uppercase ASCII letters to 2B manufacturer ID.
void	sl_rail_sdk_wmbus_frame_mfield_to_chars (const uint16_t mfield, char *manuf) Helper function for converting 2B manufacturer ID to 3 uppercase ASCII letter.
void	sl_rail_sdk_wmbus_frame_crypto5_init (void) Initializes crypto engine.
void	sl_rail_sdk_wmbus_frame_crypto5_set_key (const uint8_t *newKey) Sets key for mode5 encryption/decryption (AES-CBC)
void	sl_rail_sdk_wmbus_frame_crypto5_get_key (uint8_t *key) Returns key for mode5 encryption/decryption (AES-CBC)
uint8_t	sl_rail_sdk_wmbus_frame_crypto5_encrypt (uint8_t *input, uint8_t *output, uint8_t *iv, uint16_t length, uint8_t encryptedBlocks) Encrypts blocks using mode5 (AES-CBC) encryption.
void	sl_rail_sdk_wmbus_frame_crypto5_decrypt (uint8_t *input, uint8_t *output, uint8_t *iv, uint16_t length) Decrypts blocks using mode5 (AES-CBC) encryption.
uint16_t	sl_rail_sdk_wmbus_setup_frame (uint8_t *buffer, uint8_t access_number, sl_rail_sdk_wmbus_accessibility_t accessibility, sl_rail_sdk_wmbus_device_type_t new_device_type, sl_rail_sdk_wmbus_sensor_data_t *sensors_data, bool periodic, bool encrypt) Sets up a Wireless M-Bus frame.

Function Documentation

sl_rail_sdk_wmbus_phy_software

```
uint16_t sl_rail_sdk_wmbus_phy_software (uint8_t * buffer, uint8_t length, uint16_t buffer_length)
```

Runs the required software based phy processes on the buffer, depending on the mode.

Parameters

[inout]	buffer	The buffer that holds the data, and what will hold the processed data
[in]	length	Length of the input buffer
[in]	buffer_length	Size of the buffer, to prevent writing outside of the buffer

The buffer must be big enough to hold the processed data. Currently, only T mode's Meter to Other requires software based crc and 3of6 encoding. For that, the buffer should be at 436B long for the longest packet. All other modes passes through this call without modifications sl_rail_sdk_wmbus_set_mode should be called before using this function.

Returns

The length of the processed packet, that can be passed to the hardware, or 0 on error

Definition at line 306 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_get_accessibility

```
sl_rail_sdk_wmbus_accessibility_t sl_rail_sdk_wmbus_get_accessibility (void )
```

Get the current Wireless M-Bus accessibility settings.

Parameters

N/A		

Returns

- the current value

Definition at line 313 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_set_accessibility

```
void sl_rail_sdk_wmbus_set_accessibility (sl_rail_sdk_wmbus_accessibility_t new_wmbus_accessibility)
```

Set the current Wireless M-Bus accessibility settings.

Parameters

[in]	new_wmbus_accessibility	to be applied

Definition at line 319 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_init

```
void sl_rail_sdk_wmbus_init (void )
```

Initializes the MBUS and its mode.

Parameters

N/A		

Definition at line 324 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_set_mode

```
void sl_rail_sdk_wmbus_set_mode (sl_rail_sdk_wmbus_mode_t mode)
```

Sets the MBUS mode to use.

Parameters

[in]	mode	is the desired mode of the Wireless M-Bus

Currently only used to init and enable GPCRC for T mode Tx, and it also changes the time returned by [sl_rail_sdk_wmbus_get_meter_limited_acc_rx_start\(\)](#).

Definition at line 333 of file `/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h`

sl_rail_sdk_wmbus_get_selected_mode

`sl_rail_sdk_wmbus_mode_t sl_rail_sdk_wmbus_get_selected_mode (void )`

Gets the selected Wireless M-Bus mode.

Parameters

N/A		

Returns

- The selected mode of the Wireless M-Bus

Definition at line 340 of file `/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h`

sl_rail_sdk_wmbus_get_meter_limited_acc_rx_start

`uint32_t sl_rail_sdk_wmbus_get_meter_limited_acc_rx_start (bool slowMode)`

Returns the delay after TX when RX should start in limited accessibility `sl_rail_sdk_wmbus_set_mode` should be called before using this function.

Parameters

[in]	slowMode	True in case of slow mode. Only used in mode C and F

Returns

- The delay in microseconds

Definition at line 351 of file `/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h`

sl_rail_sdk_wmbus_get_meter_limited_acc_rx_stop

`uint32_t sl_rail_sdk_wmbus_get_meter_limited_acc_rx_stop (bool slowMode)`

Returns the delay after TX when RX should stop in limited accessibility `sl_rail_sdk_wmbus_set_mode` should be called before using this function.

Parameters

[in]	slowMode	True in case of slow mode. Only used in mode C and F

Returns

The delay in microseconds

Definition at line 362 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_frame_add_payload_int8

uint8_t sl_rail_sdk_wmbus_frame_add_payload_int8 (uint8_t * buffer, sl_rail_sdk_wmbus_frame_dib_function_t dibFunction, uint8_t vib, int8_t value)

Helper function for building the payload of a Wireless M-Bus frame.

Parameters

[in]	buffer	The buffer to write into. Should point at the first free byte
[in]	dibFunction	Function field in the DIB (Data Information Block)
[in]	vib	Value information block containing the VIF (Value Information Field)
[in]	value	The value of the data block.

Adds an 8 bit integer (binary) data block to a buffer. **Returns**

- the number of bytes written into buffer

Definition at line 373 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_frame_add_payload_int16

uint8_t sl_rail_sdk_wmbus_frame_add_payload_int16 (uint8_t * buffer, sl_rail_sdk_wmbus_frame_dib_function_t dibFunction, uint8_t vib, int16_t value)

Helper function for building the payload of a Wireless M-Bus frame.

Parameters

[in]	buffer	The buffer to write into. Should point at the first free byte
[in]	dibFunction	Function field in the DIB (Data Information Block)
[in]	vib	Value information block containing the VIF (Value Information Field)
[in]	value	The value of the data block.

Adds a 16 bit integer (binary) data block to a buffer. **Returns**

- the number of bytes written into buffer

Definition at line 385 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_frame_add_payload_int32

uint8_t sl_rail_sdk_wmbus_frame_add_payload_int32 (uint8_t * buffer, sl_rail_sdk_wmbus_frame_dib_function_t dibFunction, uint8_t vib, int32_t value)

Helper function for building the payload of a Wireless M-Bus frame.

Parameters

[in]	buffer	The buffer to write into. Should point at the first free byte
[in]	dibFunction	Function field in the DIB (Data Information Block)
[in]	vib	Value information block containing the VIF (Value Information Field)
[in]	value	The value of the data block.

Adds a 32 bit integer (binary) data block to a buffer. **Returns**

- the number of bytes written into buffer

Definition at line 397 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_frame_add_idle_filler

```
uint16_t sl_rail_sdk_wmbus_frame_add_idle_filler (uint8_t * buffer, uint16_t fillerLength)
```

Helper function for building the payload of a Wireless M-Bus frame.

Parameters

[in]	buffer	The buffer to write into. Should point at the first free byte
[in]	fillerLength	Number of bytes to fill.

Adds filler bytes required before encryption **Returns**

- the number of bytes written into buffer

Definition at line 407 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_frame_chars_to_mfield

```
uint16_t sl_rail_sdk_wmbus_frame_chars_to_mfield (const char * manuf)
```

Helper function for converting uppercase ASCII letters to 2B manufacturer ID.

Parameters

[in]	manuf	Pointer to a 3 element char array, containing the uppercase letters of the manufacturer assigned by also known as DLMS FLAG ID

Returns

- the manufacturer id (or M-field) for M-Bus addressing

Definition at line 416 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_frame_mfield_to_chars

```
void sl_rail_sdk_wmbus_frame_mfield_to_chars (const uint16_t mfield, char * manuf)
```

Helper function for converting 2B manufacturer ID to 3 uppercase ASCII letter.

Parameters

[in]	mfield	mfield value
[out]	manuf	pointer to a 3 element char array to write the 3 letters

Definition at line 424 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_frame_crypto5_init

```
void sl_rail_sdk_wmbus_frame_crypto5_init (void )
```

Initializes crypto engine.

Parameters

N/A		

Definition at line 429 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_frame_crypto5_set_key

```
void sl_rail_sdk_wmbus_frame_crypto5_set_key (const uint8_t * newKey)
```

Sets key for mode5 encryption/decryption (AES-CBC)

Parameters

[in]	newKey	16B array containing the new key

Definition at line 435 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_frame_crypto5_get_key

```
void sl_rail_sdk_wmbus_frame_crypto5_get_key (uint8_t * key)
```

Returns key for mode5 encryption/decryption (AES-CBC)

Parameters

[out]	key	Copy of the current key

Definition at line 441 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_frame_crypto5_encrypt

```
uint8_t sl_rail_sdk_wmbus_frame_crypto5_encrypt (uint8_t * input, uint8_t * output, uint8_t * iv, uint16_t length, uint8_t encryptedBlocks)
```

Encrypts blocks using mode5 (AES-CBC) encryption.

Parameters

[in]	input	Input data. Should be at least 16B*encryptedBlocks long.
[out]	output	Output data. Should be at least 16B*encryptedBlocks long. Can be the same as input.
[inout]	iv	Initialization vector, 16B long
[in]	length	Length of the input data
[in]	encryptedBlocks	Number of blocks to encode. If 0, all data will be encoded based on length

Also adds padding if needed, but the payload should include the 2B AES check (0x2f2f) in the beginning of the payload

Returns

- the number of blocks encoded. Output length is 16 times this value.

Definition at line 458 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_frame_crypto5_decrypt

```
void sl_rail_sdk_wmbus_frame_crypto5_decrypt (uint8_t * input, uint8_t * output, uint8_t * iv, uint16_t length)
```

Decrypts blocks using mode5 (AES-CBC) encryption.

Parameters

[in]	input	Input data. Should be at least length long.
[out]	output	Output data. Should be at least length long. Can be the same as input.
[inout]	iv	Initialization vector, 16B long
[in]	length	Length of the input data. Should be divisible by 16

Definition at line 470 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_setup_frame

```
uint16_t sl_rail_sdk_wmbus_setup_frame (uint8_t * buffer, uint8_t access_number, sl_rail_sdk_wmbus_accessibility_t accessibility, sl_rail_sdk_wmbus_device_type_t new_device_type, sl_rail_sdk_wmbus_sensor_data_t * sensors_data, bool periodic, bool encrypt)
```

Sets up a Wireless M-Bus frame.

Parameters

[out]	buffer	The buffer to store the Wireless M-Bus frame.
[in]	access_number	The access number of the frame.
[in]	accessibility	The accessibility of the frame.
[in]	new_device_type	The device type of the frame.
[in]	sensors_data	An array of data to be included in the frame.
[in]	periodic	Indicates whether the frame is periodic or not.
[in]	encrypt	Indicates whether the frame should be encrypted or not.

Sets up a Wireless M-Bus frame by populating the provided buffer with the necessary information.

Returns

- The length of the assembled frame.

Definition at line 93 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_packet_assembler/sl_rail_sdk_wmbus_packet_assembler.h

Configurations

Configurations

Macros

```
#define SL_RAIL_SDK_WMBUS_TYPE WMBUS_MODE_T_METER
Default Wireless M-Bus mode.

#define SL_RAIL_SDK_WMBUS_ACCESSIBILITY WMBUS_ACCESSIBILITY_LIMITED_ACCESS
Default Wireless M-Bus Accessibility.
```

Type definitions

Type definitions

Modules

[sl_rail_sdk_wmbus_dll_header_t](#)

[sl_rail_sdk_wmbus_conf_word_t](#)

[sl_rail_sdk_wmbus_stl_header_t](#)

[sl_rail_sdk_wmbus_ltl_header_t](#)

[sl_rail_sdk_wmbus_sensor_data_t](#)

Enumerations

```
enum    sl_rail_sdk_wmbus_function_code_t {
    WMBUS_FUNCTION_SND_NKE = 0x0
    WMBUS_FUNCTION_SND_UD = 0x3
    WMBUS_FUNCTION_SND_NR = 0x4
    WMBUS_FUNCTION_SND_IR = 0x6
    WMBUS_FUNCTION_ACC_NR = 0x7
    WMBUS_FUNCTION_ACC_DMD = 0x8
    WMBUS_FUNCTION_REQ_UD1 = 0xA
    WMBUS_FUNCTION_REQ_UD2 = 0xB
}
Function code to use in the first block's C-field.
```

```
enum    sl_rail_sdk_wmbus_device_type_t {
    WMBUS_DEVICE_OTHER = 0x00
    WMBUS_DEVICE_OIL_METER
    WMBUS_DEVICE_ELECTRICITY_METER
    WMBUS_DEVICE_GAS_METER
    WMBUS_DEVICE_HEAT_METER
    WMBUS_DEVICE_STEAM_METER
    WMBUS_DEVICE_WARM_WATER_METER
    WMBUS_DEVICE_WATER_METER
    WMBUS_DEVICE_HEAT_COST_ALLOCATOR
    WMBUS_DEVICE_COMPRESSED_AIR
    WMBUS_DEVICE_COOLING_METER_OUTLET
    WMBUS_DEVICE_COOLING_METER_INLET
    WMBUS_DEVICE_HEAT_METER_INLET
    WMBUS_DEVICE_COMBINED_HEAT_COOLING_METER
    WMBUS_DEVICE_BUS_SYSTEM_COMPONENT
    WMBUS_DEVICE_CALORIFIC_VALUE = 0x14
    WMBUS_DEVICE_HOT_WATER_METER
    WMBUS_DEVICE_COLD_WATER_METER
    WMBUS_DEVICE_DUAL_WATER_METER
    WMBUS_DEVICE_PRESSURE_METER
    WMBUS_DEVICE_AD_CONVERTER
    WMBUS_DEVICE_SMOKE_DETECTOR
    WMBUS_DEVICE_ROOM_SENSOR
    WMBUS_DEVICE_GAS_DETECTOR
    WMBUS_DEVICE_BREAKER = 0x20
    WMBUS_DEVICE_VALVE
    WMBUS_DEVICE_CUSTOMER_UNIT = 0x25
    WMBUS_DEVICE_WASTE_WATER_METER = 0x28
    WMBUS_DEVICE_GARBAGE
```



```

WMBUS_DEVICE CARBON_DIOXIDE
WMBUS_DEVICE COMMUNICATION_CONTROLLER
= 0x31
WMBUS_DEVICE UNIDIRECTIONAL_REPEATER
WMBUS_DEVICE BIDIRECTIONAL_REPEATER
WMBUS_DEVICE RADIO_CONVERTER_SYSTEM =
0x36
WMBUS_DEVICE RADIO_CONVERTER_METER

```

```

}

```

Device type to be used in the address field(s)

```

enum    sl_rail_sdk_wmbus_ci_field_t {

    WMBUS_CI_DATA_TO_METER_NONE = 0x51
    WMBUS_CI_MBUS_DATA_TO_METER_SHORT = 0x5A
    WMBUS_CI_MBUS_DATA_TO_METER_LONG
    WMBUS_CI_COSEM_DATA_TO_METER_LONG = 0x60
    WMBUS_CI_COSEM_DATA_TO_METER_SHORT
    WMBUS_CI_OBIS_DATA_TO_METER_LONG = 0x64
    WMBUS_CI_OBIS_DATA_TO_METER_SHORT
    WMBUS_CI_EN13757.3_DATA_TO_METER_NONE = 0x69
    WMBUS_CI_EN13757.3_DATA_TO_METER_SHORT
    WMBUS_CI_EN13757.3_DATA_TO_METER_LONG
    WMBUS_CI_CLOCK_SYNC_ABSOLUTE
    WMBUS_CI_CLOCK_SYNC_RELATIVE
    WMBUS_CI_APPLICATION_ERROR_SHORT
    WMBUS_CI_APPLICATION_ERROR_LONG
    WMBUS_CI_APPLICATION_ERROR_NONE
    WMBUS_CI_ALARM_NONE
    WMBUS_CI_EN13757.3_APPLICATION_LONG
    WMBUS_CI_EN13757.3_APPLICATION_CF_LONG
    WMBUS_CI_ALARM_SHORT
    WMBUS_CI_ALARM_LONG
    WMBUS_CI_EN13757.3_APPLICATION_NONE = 0x78
    WMBUS_CI_EN13757.3_APPLICATION_CF_NONE
    WMBUS_CI_EN13757.3_APPLICATION_SHORT
    WMBUS_CI_EN13757.3_APPLICATION_CF_SHORT
    WMBUS_CI_COSEM_APPLICATION_LONG
    WMBUS_CI_COSEM_APPLICATION_SHORT
    WMBUS_CI_OBIS_APPLICATION_LONG
    WMBUS_CI_OBIS_APPLICATION_SHORT
    WMBUS_CI_EN13757.3_TRANSPORT_TO_METER_LONG
    WMBUS_CI_NETWORK_LAYER_DATA
    WMBUS_CI_FOR_FUTURE_USE
    WMBUS_CI_NETWORK_MANAGEMENT_APPLICATION
    WMBUS_CI_TRANSPORT_TO_METER_MBUS_COMPACT
    WMBUS_CI_NETWORK_MANAGEMENT_DATA = 0x89
    WMBUS_CI_EN13757.3_TRANSPORT_FROM_METER_SHORT
    WMBUS_CI_EN13757.3_TRANSPORT_FROM_METER_LONG
    WMBUS_CI_EXTENDED_LINK_LAYER1
    WMBUS_CI_EXTENDED_LINK_LAYER2
    WMBUS_CI_EXTENDED_LINK_LAYER3
    WMBUS_CI_EXTENDED_LINK_LAYER4

```

```

}

```

CI field codes.

```

enum    sl_rail_sdk_wmbus_status_t {

    WMBUS_STATUS_APPLICATION_BUSY = 0x1
    WMBUS_STATUS_APPLICATION_ERROR = 0x2
    WMBUS_STATUS_ABNORMAL_CONDITION = 0x3
    WMBUS_STATUS_POWER_LOW = 1 << 2
    WMBUS_STATUS_PERMANENT_ERROR = 1 << 3
    WMBUS_STATUS_TEMPORARY_ERROR = 1 << 4

```

```

}

```

Application errors coded into the status field.

```
enum sl_rail_sdk_wmbus_accessibility_t {
    WMBUS_ACCESSIBILITY_NO_ACCESS = 0
    WMBUS_ACCESSIBILITY_TEMPORARY_NO_ACCESS
    WMBUS_ACCESSIBILITY_LIMITED_ACCESS
    WMBUS_ACCESSIBILITY_UNLIMITED_ACCESS
}
Accessibility settings in the config word.

enum sl_rail_sdk_wmbus_frame_dib_function_t {
    WMBUS_DIB_FUNCTION_INSTANTANEOUS
    WMBUS_DIB_FUNCTION_MINIMUM
    WMBUS_DIB_FUNCTION_MAXIMUM
    WMBUS_DIB_FUNCTION_ERROR
}
Values for function field in DIB.

enum sl_rail_sdk_wmbus_mode_t {
    WMBUS_MODE_S
    WMBUS_MODE_T_METER
    WMBUS_MODE_T_COLLECTOR
    WMBUS_MODE_C
    WMBUS_MODE_F
    WMBUS_MODE_R2
    WMBUS_MODE_N_24
    WMBUS_MODE_N_48
    WMBUS_MODE_N_192
}
MBUS modes.
```

Enumeration Documentation

sl_rail_sdk_wmbus_function_code_t

```
sl_rail_sdk_wmbus_function_code_t
```

Function code to use in the first block's C-field.

Enumerator	
WMBUS_FUNCTION_SND_NKE	SND-NKE.
WMBUS_FUNCTION_SND_UD	SND-UD/SND-UD2.
WMBUS_FUNCTION_SND_NR	SND-NR.
WMBUS_FUNCTION_SND_IR	SND-IR.
WMBUS_FUNCTION_ACC_NR	ACC-NR.
WMBUS_FUNCTION_ACC_DMD	ACC-DMD.
WMBUS_FUNCTION_REQ_UD1	REQ-UD1.
WMBUS_FUNCTION_REQ_UD2	REQ-UD2.

Definition at line 56 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_device_type_t

```
sl_rail_sdk_wmbus_device_type_t
```

Device type to be used in the address field(s)

Enumerator	
WMBUS_DEVICE_OTHER	Other.
WMBUS_DEVICE_OIL_METER	Oil meter.
WMBUS_DEVICE_ELECTRICITY_METER	Electricity meter.
WMBUS_DEVICE_GAS_METER	Gas meter.
WMBUS_DEVICE_HEAT_METER	Heat meter.
WMBUS_DEVICE_STEAM_METER	Steam meter.
WMBUS_DEVICE_WARM_WATER_METER	Warm water meter (30-90C)
WMBUS_DEVICE_WATER_METER	Water meter.
WMBUS_DEVICE_HEAT_COST_ALLOCATOR	Heat cost allocator.
WMBUS_DEVICE_COMPRESSED_AIR	Compressed air.
WMBUS_DEVICE_COOLING_METER_OUTLET	Cooling meter (volume measured at return temperature: outlet)
WMBUS_DEVICE_COOLING_METER_INLET	Cooling meter (volume measured at flow temperature: inlet)
WMBUS_DEVICE_HEAT_METER_INLET	Heat meter (volume measured at flow temperature: inlet)
WMBUS_DEVICE_COMBINED_HEAT_COOLING_METER	Combined heat/cooling meter.
WMBUS_DEVICE_BUS_SYSTEM_COMPONENT	Bus/system component.
WMBUS_DEVICE_CALORIFIC_VALUE	Calorific value.
WMBUS_DEVICE_HOT_WATER_METER	Hot water meter (>90C)
WMBUS_DEVICE_COLD_WATER_METER	Cold water meter.
WMBUS_DEVICE_DUAL_WATER_METER	Dual register (hot/cold) water meter.
WMBUS_DEVICE_PRESSURE_METER	Pressure meter.
WMBUS_DEVICE_AD_CONVERTER	A/D converter.
WMBUS_DEVICE_SMOKE_DETECTOR	Smoke detector.
WMBUS_DEVICE_ROOM_SENSOR	Room sensor (e.g.
WMBUS_DEVICE_GAS_DETECTOR	Gas detector.
WMBUS_DEVICE_BREAKER	Breaker (electricity)
WMBUS_DEVICE_VALVE	Valve (gas or water)
WMBUS_DEVICE_CUSTOMER_UNIT	Customer unit (display device)
WMBUS_DEVICE_WASTE_WATER_METER	Waste water meter.
WMBUS_DEVICE_GARBAGE	Garbage.
WMBUS_DEVICE CARBON_DIOXIDE	Reserved for carbon dioxide.
WMBUS_DEVICE_COMMUNICATION_CONTROLLER	Communication controller (Gateway)
WMBUS_DEVICE_UNIDIRECTIONAL_REPEATER	Unidirectional repeater.
WMBUS_DEVICE_BIDIRECTIONAL_REPEATER	Bidirectional repeater.
WMBUS_DEVICE_RADIO_CONVERTER_SYSTEM	Radio converter (system side)
WMBUS_DEVICE_RADIO_CONVERTER_METER	Radio converter (meter side)

Definition at line 71 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_ci_field_t

sl_rail_sdk_wmbus_ci_field_t

CI field codes.

	Enumerator
WMBUS_CI_DATA_TO_METER_NONE	Data sent by readout device to the Meter without transport layer.
WMBUS_CI_MBUS_DATA_TO_METER_SHORT	MBUS data sent by readout device to the Meter with short transport layer.
WMBUS_CI_MBUS_DATA_TO_METER_LONG	MBUS data sent by readout device to the Meter with long transport layer.
WMBUS_CI_COSEM_DATA_TO_METER_LONG	COSEM data sent by readout device to the Meter with long transport layer.
WMBUS_CI_COSEM_DATA_TO_METER_SHORT	COSEM data sent by readout device to the Meter with short transport layer.
WMBUS_CI_OBIS_DATA_TO_METER_LONG	OBIS based data sent by readout device to the Meter with long transport layer.
WMBUS_CI_OBIS_DATA_TO_METER_SHORT	OBIS based data sent by readout device to the Meter with short transport layer.
WMBUS_CI_EN13757_3_DATA_TO_METER_NONE	EN13757-3 application layer with Format frame and no transport layer.
WMBUS_CI_EN13757_3_DATA_TO_METER_SHORT	EN13757-3 application layer with Format frame and short transport layer.
WMBUS_CI_EN13757_3_DATA_TO_METER_LONG	EN13757-3 application layer with Format frame and long transport layer.
WMBUS_CI_CLOCK_SYNC_ABSOLUTE	Clock synchronisation (absolute)
WMBUS_CI_CLOCK_SYNC_RELATIVE	Clock synchronisation (relative)
WMBUS_CI_APPLICATION_ERROR_SHORT	Application error from Meter with short transport layer.
WMBUS_CI_APPLICATION_ERROR_LONG	Application error from Meter with long transport layer.
WMBUS_CI_APPLICATION_ERROR_NONE	Application error from Meter with no transport layer.
WMBUS_CI_ALARM_NONE	Alarm from Meter without transport layer.
WMBUS_CI_EN13757_3_APPLICATION_LONG	EN13757-3 application layer with long transport layer.
WMBUS_CI_EN13757_3_APPLICATION_CF_LONG	EN13757-3 application layer with compact frame and long transport layer.
WMBUS_CI_ALARM_SHORT	Alarm from Meter with short transport layer.
WMBUS_CI_ALARM_LONG	Alarm from Meter with long transport layer.
WMBUS_CI_EN13757_3_APPLICATION_NONE	EN13757-3 application layer with no transport layer.
WMBUS_CI_EN13757_3_APPLICATION_CF_NONE	EN13757-3 application layer with compact frame and no transport layer.
WMBUS_CI_EN13757_3_APPLICATION_SHORT	EN13757-3 application layer with short transport layer.
WMBUS_CI_EN13757_3_APPLICATION_CF_SHORT	EN13757-3 application layer with compact frame and short transport layer.
WMBUS_CI_COSEM_APPLICATION_LONG	COSEM application layer with long transport layer.
WMBUS_CI_COSEM_APPLICATION_SHORT	COSEM application layer with short transport layer.
WMBUS_CI_OBIS_APPLICATION_LONG	OBIS based application layer with long transport layer.
WMBUS_CI_OBIS_APPLICATION_SHORT	OBIS based application layer with short transport layer.
WMBUS_CI_EN13757_3_TRANSPORT_TO_METER_LONG	EN13757-3 Transport layer (long) from readout device to the meter.
WMBUS_CI_NETWORK_LAYER_DATA	Network layer data.

WMBUS_CI_FOR_FUTURE_USE	For future use.
WMBUS_CI_NETWORK_MANAGEMENT_APPLICATION	Network management application.
WMBUS_CI_TRANSPORT_TO_METER_MBUS_COMPACT	Transport layer to Meter (M Bus-Compact frame expected)
WMBUS_CI_NETWORK_MANAGEMENT_DATA	Reserved for Network management data (EN 13757-5)
WMBUS_CI_EN13757_3_TRANSPORT_FROM_METER_SHORT	EN13757-3 Transport layer (short) from the meter to the readout device.
WMBUS_CI_EN13757_3_TRANSPORT_FROM_METER_LONG	EN13757-3 Transport layer (long) from the meter to the readout device.
WMBUS_CI_EXTENDED_LINK_LAYER1	Extended Link Layer I (2B)
WMBUS_CI_EXTENDED_LINK_LAYER2	Extended Link Layer II (8B)
WMBUS_CI_EXTENDED_LINK_LAYER3	Extended Link Layer III (10B)
WMBUS_CI_EXTENDED_LINK_LAYER4	Extended Link Layer IV (16B)

Definition at line 113 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_status_t

sl_rail_sdk_wmbus_status_t

Application errors coded into the status field.

Enumerator	
WMBUS_STATUS_APPLICATION_BUSY	Application busy.
WMBUS_STATUS_APPLICATION_ERROR	Any application error.
WMBUS_STATUS_ABNORMAL_CONDITION	Abnormal condition/alarm.
WMBUS_STATUS_POWER_LOW	Power low.
WMBUS_STATUS_PERMANENT_ERROR	Permanent error.
WMBUS_STATUS_TEMPORARY_ERROR	Temporary error.

Definition at line 160 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_accessibility_t

sl_rail_sdk_wmbus_accessibility_t

Accessibility settings in the config word.

Enumerator	
WMBUS_ACCESSIBILITY_NO_ACCESS	No access (unidirectional)
WMBUS_ACCESSIBILITY_TEMPORARY_NO_ACCESS	Bidirectional, temporary no access.
WMBUS_ACCESSIBILITY_LIMITED_ACCESS	Bidirectional, limited access (short time after TX)
WMBUS_ACCESSIBILITY_UNLIMITED_ACCESS	Bidirectional, unlimited access.

Definition at line 173 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_frame_dib_function_t

sl_rail_sdk_wmbus_frame_dib_function_t

Values for function field in DIB.

Enumerator	
WMBUS_DIB_FUNCTION_INSTANTANEOUS	Instantaneous value.
WMBUS_DIB_FUNCTION_MINIMUM	Minimum value.
WMBUS_DIB_FUNCTION_MAXIMUM	Maximum value.
WMBUS_DIB_FUNCTION_ERROR	Value during error state.

Definition at line 267 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_mode_t

sl_rail_sdk_wmbus_mode_t

MBUS modes.

Enumerator	
WMBUS_MODE_S	S mode.
WMBUS_MODE_T_METER	T mode, Meter to Other.
WMBUS_MODE_T_COLLECTOR	T mode, Other to Meter.
WMBUS_MODE_C	C mode.
WMBUS_MODE_F	F mode.
WMBUS_MODE_R2	R2 mode.
WMBUS_MODE_N_24	N mode, 2.4kbps (channel 2a and 2b)
WMBUS_MODE_N_48	N mode, 4.8kbps (channel 1a, 1b, 3a and 3b)
WMBUS_MODE_N_192	N mode, 19.2kbps (channel 0)

Definition at line 278 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_dll_header_t

Data link layer header (first block)

Public Attributes

uint8_t	IField Length field.
uint8_t	raw Control field raw mode.
sl_rail_sdk_wmbus_function_code_t	function_code Control field function code.
bool	fcv_dcf Control field fcv/dcf.
bool	fcb_acd Control field fcb/acd.
bool	primary Control field, primary station bit.
struct sl_rail_sdk_wmbus_dll_header_t::@0::@2	detailed Control field detailed mode.
union sl_rail_sdk_wmbus_dll_header_t::@0	c_field Control field.
uint16_t	manufacturer Address, manufacturer ID.
uint32_t	id Address, device ID.
uint8_t	version Address, version.
uint8_t	device_type Address, device type.
struct sl_rail_sdk_wmbus_dll_header_t::@1::@3	detailed Address field detailed mode.
union sl_rail_sdk_wmbus_dll_header_t::@1	address Address field.

Public Attribute Documentation

IField

uint8_t sl_rail_sdk_wmbus_dll_header_t::IField

Length field.

Definition at line 189 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

raw

```
uint8_t sl_rail_sdk_wmbus_dll_header_t::raw[8]
```

Control field raw mode.

Address field raw mode.

Definition at line 191 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

function_code

```
sl_rail_sdk_wmbus_function_code_t sl_rail_sdk_wmbus_dll_header_t::function_code
```

Control field function code.

Definition at line 193 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

fcv_dcf

```
bool sl_rail_sdk_wmbus_dll_header_t::fcv_dcf
```

Control field fcv/dcf.

Definition at line 194 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

fcv_acd

```
bool sl_rail_sdk_wmbus_dll_header_t::fcv_acd
```

Control field fcb/acd.

Definition at line 195 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

primary

```
bool sl_rail_sdk_wmbus_dll_header_t::primary
```


Control field, primary station bit.

Definition at line 196 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

detailed

```
struct sl_rail_sdk_wmbus_dll_header_t::@0::@2 sl_rail_sdk_wmbus_dll_header_t::detailed
```

Control field detailed mode.

Definition at line 197 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

c_field

```
union sl_rail_sdk_wmbus_dll_header_t::@0 sl_rail_sdk_wmbus_dll_header_t::c_field
```

Control field.

Definition at line 198 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

manufacturer

```
uint16_t sl_rail_sdk_wmbus_dll_header_t::manufacturer
```

Address, manufacturer ID.

Definition at line 202 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

id

```
uint32_t sl_rail_sdk_wmbus_dll_header_t::id
```

Address, device ID.

Definition at line 203 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

version

```
uint8_t sl_rail_sdk_wmbus_dll_header_t::version
```

Address, version.

Definition at line 204 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

device_type

```
uint8_t sl_rail_sdk_wmbus_dll_header_t::device_type
```

Address, device type.

Definition at line 205 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

detailed

```
struct sl_rail_sdk_wmbus_dll_header_t::@1::@3 sl_rail_sdk_wmbus_dll_header_t::detailed
```

Address field detailed mode.

Definition at line 206 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

address

```
union sl_rail_sdk_wmbus_dll_header_t::@1 sl_rail_sdk_wmbus_dll_header_t::address
```

Address field.

Definition at line 207 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_conf_word_t

Data link layer header (first block)

Public Attributes

uint16_t	raw	Raw config word.
bool	hop_counter	Hopcounter.
bool	repeated_access	Repeated access.
uint8_t	content_of_message	Content of message.
uint8_t	num_of_enc_blocks	Number of encoded blocks.
uint8_t	mode	Mode (security) 5th bit is reserved as per 13757-4, but OMS uses it.
bool	synchronized	Synchronized or not.
sl_rail_sdk_wmbus_accessibility_t	accessibility	Accessibility.
struct sl_rail_sdk_wmbus_conf_word_t::@4	mode_0_5	Detailed config word for mode 0 and 5.

Public Attribute Documentation

raw

uint16_t sl_rail_sdk_wmbus_conf_word_t::raw

Raw config word.

Definition at line 215 of file `/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h`

hop_counter

bool sl_rail_sdk_wmbus_conf_word_t::hop_counter

Hopcounter.

Definition at line 217 of file `/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h`

repeated_access

```
bool sl_rail_sdk_wmbus_conf_word_t::repeated_access
```

Repeated access.

Definition at line 218 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

content_of_message

```
uint8_t sl_rail_sdk_wmbus_conf_word_t::content_of_message
```

Content of message.

Definition at line 219 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

num_of_enc_blocks

```
uint8_t sl_rail_sdk_wmbus_conf_word_t::num_of_enc_blocks
```

Number of encoded blocks.

Definition at line 220 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

mode

```
uint8_t sl_rail_sdk_wmbus_conf_word_t::mode
```

Mode (security) 5th bit is reserved as per 13757-4, but OMS uses it.

Definition at line 221 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

synchronized

```
bool sl_rail_sdk_wmbus_conf_word_t::synchronized
```

Synchronized or not.

Definition at line 222 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

accessibility

```
sl_rail_sdk_wmbus_accessibility_t sl_rail_sdk_wmbus_conf_word_t::accessibility
```

Accessibility.

Definition at line 223 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

mode_0_5

```
struct sl_rail_sdk_wmbus_conf_word_t::@4 sl_rail_sdk_wmbus_conf_word_t::mode_0_5
```

Detailed config word for mode 0 and 5.

Definition at line 224 of file

/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_stl_header_t

Short transport layer header.

Public Attributes

uint8_t	ci_field	CI (control information) field.
uint8_t	access_number	Access number.
uint8_t	status	Status byte.
sl_rail_sdk_wmbus_conf_word_t	conf_word	Config word.

Public Attribute Documentation

ci_field

```
uint8_t sl_rail_sdk_wmbus_stl_header_t::ci_field
```

CI (control information) field.

Definition at line 233 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

access_number

```
uint8_t sl_rail_sdk_wmbus_stl_header_t::access_number
```

Access number.

Definition at line 234 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

status

```
uint8_t sl_rail_sdk_wmbus_stl_header_t::status
```

Status byte.

Definition at line 235 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

conf_word

```
sl_rail_sdk_wmbus_conf_word_t sl_rail_sdk_wmbus_stl_header_t::conf_word
```

Config word.

Definition at line 236 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_ltl_header_t

Long transport layer header.

Public Attributes

uint8_t	ci_field	CI (control information) field.
uint8_t	raw	Raw address.
uint32_t	id	Address, device ID.
uint16_t	manufacturer	Address, manufacturer ID.
uint8_t	version	Address, version.
uint8_t	device_type	Address, device type.
struct sl_rail_sdk_wmbus _ltl_header_t::@5:: @6	detailed	Detailed address.
union sl_rail_sdk_wmbus _ltl_header_t::@5	address	Address.
uint8_t	access_number	Access number.
uint8_t	status	Status byte.
sl_rail_sdk_wmbus _conf_word_t	conf_word	Config word.

Public Attribute Documentation

ci_field

uint8_t sl_rail_sdk_wmbus_ltl_header_t::ci_field

CI (control information) field.

Definition at line 244 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

raw

uint8_t sl_rail_sdk_wmbus_ltl_header_t::raw[8]

Raw address.

Definition at line 246 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

id

uint32_t sl_rail_sdk_wmbus_itl_header_t::id

Address, device ID.

Definition at line 248 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

manufacturer

uint16_t sl_rail_sdk_wmbus_itl_header_t::manufacturer

Address, manufacturer ID.

Definition at line 249 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

version

uint8_t sl_rail_sdk_wmbus_itl_header_t::version

Address, version.

Definition at line 250 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

device_type

uint8_t sl_rail_sdk_wmbus_itl_header_t::device_type

Address, device type.

Definition at line 251 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

detailed

struct sl_rail_sdk_wmbus_itl_header_t::@5::@6 sl_rail_sdk_wmbus_itl_header_t::detailed

Detailed address.

Definition at line 252 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

address

```
union sl_rail_sdk_wmbus_ltl_header_t:@5 sl_rail_sdk_wmbus_ltl_header_t::address
```

Address.

Definition at line 253 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

access_number

```
uint8_t sl_rail_sdk_wmbus_ltl_header_t::access_number
```

Access number.

Definition at line 254 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

status

```
uint8_t sl_rail_sdk_wmbus_ltl_header_t::status
```

Status byte.

Definition at line 255 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

conf_word

```
sl_rail_sdk_wmbus_conf_word_t sl_rail_sdk_wmbus_ltl_header_t::conf_word
```

Config word.

Definition at line 256 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_support/sl_rail_sdk_wmbus_support.h

sl_rail_sdk_wmbus_sensor_data_t

Structure representing a Wireless M-Bus sensor data.

Contains the value information field and the data field.

Public Attributes

uint32_t	vif
uint32_t	data
struct sl_rail_sdk_wmbus_sensor_data *	next

Public Attribute Documentation

vif

```
uint32_t sl_rail_sdk_wmbus_sensor_data_t::vif
```

value information field

Definition at line 64 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_packet_assembler/sl_rail_sdk_wmbus_packet_assembler.h

data

```
uint32_t sl_rail_sdk_wmbus_sensor_data_t::data
```

data field

Definition at line 65 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_packet_assembler/sl_rail_sdk_wmbus_packet_assembler.h

next

```
struct sl_rail_sdk_wmbus_sensor_data* sl_rail_sdk_wmbus_sensor_data_t::next
```

pointer to the next item

Definition at line 66 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_packet_assembler/sl_rail_sdk_wmbus_packet_assembler.h

Wireless M-Bus Thermometer Sensor

Wireless M-Bus Thermometer Sensor

The Wireless M-Bus Thermometer Sensor component is designed to monitor and measure temperature using a thermometer sensor. This component provides the necessary tools to integrate temperature data into Wireless M-Bus packets, ensuring that the data is correctly formatted and transmitted.

Key Features:

- **Temperature Measurement:** Functions to measure temperature using a thermometer sensor.
- **Data Handling:** Functions to retrieve and format temperature data for inclusion in Wireless M-Bus packets.
- **Packet Assembly:** Tools to assemble packets with temperature data, ensuring that the data is correctly formatted and transmitted.

Modules

[Configurations](#)

Functions

sl_status_t	sl_rail_sdk_wmbus_sensor_thermometer_init (void) Wireless M-Bus Thermometer initialization function.
sl_status_t	sl_rail_sdk_wmbus_sensor_thermometer_deinit (void) Wireless M-Bus Thermometer deinitialization function.
sl_status_t	sl_rail_sdk_wmbus_sensor_thermometer_measure (void) Wireless M-Bus Thermometer measure function function.
sl_rail_sdk_wmbus_sensor_data_t *	sl_rail_sdk_wmbus_sensor_thermometer_get_data (void) Retrieves the sensor data for the Wireless M-Bus thermo meter sensor.
sl_status_t	sl_rail_sdk_wmbus_sensor_thermometer_print (void) Wireless M-Bus Thermometer 7 segment LCD print function.
sl_status_t	sl_rail_sdk_wmbus_sensor_thermometer_button_pressed (void) Wireless M-Bus Thermometer button pressed function.

Function Documentation

sl_rail_sdk_wmbus_sensor_thermometer_init

```
sl_status_t sl_rail_sdk_wmbus_sensor_thermometer_init (void )
```

Wireless M-Bus Thermometer initialization function.

Parameters

N/A		

Returns

- sl_status_t: SL_STATUS_OK if init was successful SL_STATUS_FAIL if init failed

Definition at line 67 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_thermometer/sl_rail_sdk_wmbus_sensor_thermometer.h

sl_rail_sdk_wmbus_sensor_thermometer_deinit

```
sl_status_t sl_rail_sdk_wmbus_sensor_thermometer_deinit (void )
```

Wireless M-Bus Thermometer deinitialization function.

Parameters

N/A		

Returns

- sl_status_t: SL_STATUS_OK if init was successful SL_STATUS_FAIL if init failed

Definition at line 75 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_thermometer/sl_rail_sdk_wmbus_sensor_thermometer.h

sl_rail_sdk_wmbus_sensor_thermometer_measure

```
sl_status_t sl_rail_sdk_wmbus_sensor_thermometer_measure (void )
```

Wireless M-Bus Thermometer measure function function.

Parameters

N/A		

Read sensor data.

Returns

- sl_status_t: SL_STATUS_OK if measure was successful SL_STATUS_FAIL if measure failed

Definition at line 83 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_thermometer/sl_rail_sdk_wmbus_sensor_thermometer.h

sl_rail_sdk_wmbus_sensor_thermometer_get_data

```
sl_rail_sdk_wmbus_sensor_data_t * sl_rail_sdk_wmbus_sensor_thermometer_get_data (void )
```

Retrieves the sensor data for the Wireless M-Bus thermo meter sensor.

Parameters

N/A		

Returns a pointer to the [sl_rail_sdk_wmbus_sensor_data_t](#) structure that contains the sensor data for the Wireless M-Bus thermo meter sensor.

Returns

- Pointer to a [sl_rail_sdk_wmbus_sensor_data_t](#), the sensor data for the Wireless M-Bus thermo meter sensor

Definition at line 94 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_thermometer/sl_rail_sdk_wmbus_sensor_thermometer.h

sl_rail_sdk_wmbus_sensor_thermometer_print

```
sl_status_t sl_rail_sdk_wmbus_sensor_thermometer_print (void )
```

Wireless M-Bus Thermometer 7 segment LCD print function.

Parameters

N/A		

Returns

- sl_status_t: SL_STATUS_OK if measure was successful SL_STATUS_FAIL if measure failed

Definition at line 102 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_thermometer/sl_rail_sdk_wmbus_sensor_thermometer.h

sl_rail_sdk_wmbus_sensor_thermometer_button_pressed

```
sl_status_t sl_rail_sdk_wmbus_sensor_thermometer_button_pressed (void )
```

Wireless M-Bus Thermometer button pressed function.

Parameters

N/A		

This function handles the button press event for the Wireless M-Bus thermo meter sensor, in this case, it changes the displayed value between temperature and humidity.

Returns

- sl_status_t: SL_STATUS_OK if measure was successful SL_STATUS_FAIL if measure failed

Definition at line 112 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_thermometer/sl_rail_sdk_wmbus_sensor_thermometer.h

Configurations

Configurations

Macros

```
#define SL_RAIL_SDK_WMBUS_SENSOR_THERMOMETER_SLEEPTIMER_PRIORITY 10
Default sleeptimer priority.

#define SL_RAIL_SDK_WMBUS_SENSOR_THERMOMETER_SLEEPTIMER_TIMEOUT 1000
Default sleeptimer timeout.
```

Wireless M-Bus Virtual Water Meter Sensor

Wireless M-Bus Virtual Water Meter Sensor

The Wireless M-Bus Virtual Water Meter Sensor component is a software-based solution designed to monitor and measure water usage in a given environment. This component provides the necessary tools to integrate water meter data into Wireless M-Bus packets, ensuring that the data is correctly formatted and transmitted.

Key Features:

- **Water Usage Measurement:** Functions to measure water usage using a virtual water meter sensor.
- **Data Handling:** Functions to retrieve and format water meter data for inclusion in Wireless M-Bus packets.
- **Packet Assembly:** Tools to assemble packets with water meter data, ensuring that the data is correctly formatted and transmitted.

Modules

[Configurations](#)

Functions

sl_status_t	sl_rail_sdk_wmbus_sensor_virtual_water_meter_init(void) Wireless M-Bus Virtual Water Meter initialization function.
sl_status_t	sl_rail_sdk_wmbus_sensor_virtual_water_meter_deinit(void) Wireless M-Bus Virtual Water Meter deinitialization function.
sl_status_t	sl_rail_sdk_wmbus_sensor_virtual_water_meter_measure(void) Wireless M-Bus Virtual Water Meter measure function.
sl_rail_sdk_wmbus_sensor_data_t *	sl_rail_sdk_wmbus_sensor_virtual_water_meter_get_data(void) Retrieves the sensor data for the Wireless M-Bus virtual water meter sensor.
sl_status_t	sl_rail_sdk_wmbus_sensor_virtual_water_meter_print(void) Wireless M-Bus Virtual Water Meter 7 segment LCD print function.
sl_status_t	sl_rail_sdk_wmbus_sensor_virtual_water_meter_button_pressed(void) Wireless M-Bus Virtual Water Meter button pressed function.

Function Documentation

sl_rail_sdk_wmbus_sensor_virtual_water_meter_init

sl_status_t sl_rail_sdk_wmbus_sensor_virtual_water_meter_init (void)

Wireless M-Bus Virtual Water Meter initialization function.

Parameters

N/A		

Returns

- sl_status_t: SL_STATUS_OK if init was successful SL_STATUS_FAIL if init failed

Definition at line 67 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_virtual_water_meter/sl_rail_sdk_wmbus_sensor_virtual_water_meter.h

sl_rail_sdk_wmbus_sensor_virtual_water_meter_deinit

```
sl_status_t sl_rail_sdk_wmbus_sensor_virtual_water_meter_deinit (void )
```

Wireless M-Bus Virtual Water Meter deinitialization function.

Parameters

N/A		

- Returns
- sl_status_t: SL_STATUS_OK if init was successful SL_STATUS_FAIL if init failed

Definition at line 75 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_virtual_water_meter/sl_rail_sdk_wmbus_sensor_virtual_water_meter.h

sl_rail_sdk_wmbus_sensor_virtual_water_meter_measure

```
sl_status_t sl_rail_sdk_wmbus_sensor_virtual_water_meter_measure (void )
```

Wireless M-Bus Virtual Water Meter measure function.

Parameters

N/A		

- Returns
- sl_status_t: SL_STATUS_OK if measure was successful SL_STATUS_FAIL if measure failed

Definition at line 83 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_virtual_water_meter/sl_rail_sdk_wmbus_sensor_virtual_water_meter.h

sl_rail_sdk_wmbus_sensor_virtual_water_meter_get_data

```
sl_rail_sdk_wmbus_sensor_data_t * sl_rail_sdk_wmbus_sensor_virtual_water_meter_get_data (void )
```

Retrieves the sensor data for the Wireless M-Bus virtual water meter sensor.

Parameters

N/A		

Returns a pointer to the [sl_rail_sdk_wmbus_sensor_data_t](#) structure that contains the sensor data for the Wireless M-Bus virtual water meter sensor.

- Returns
- Pointer of a [sl_rail_sdk_wmbus_sensor_data_t](#), to the sensor data for the Wireless M-Bus virtual water meter sensor

Definition at line 94 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_virtual_water_meter/sl_rail_sdk_wmbus_sensor_virtual

water_meter.h

sl_rail_sdk_wmbus_sensor_virtual_water_meter_print

sl_status_t sl_rail_sdk_wmbus_sensor_virtual_water_meter_print (void)

Wireless M-Bus Virtual Water Meter 7 segment LCD print function.

Parameters

N/A		

Returns

- sl_status_t: SL_STATUS_OK if 7 segment LCD print was successfully SL_STATUS_FAIL if 7 segment LCD print failed

Definition at line 102 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_virtual_water_meter/sl_rail_sdk_wmbus_sensor_virtual_water_meter.h

sl_rail_sdk_wmbus_sensor_virtual_water_meter_button_pressed

sl_status_t sl_rail_sdk_wmbus_sensor_virtual_water_meter_button_pressed (void)

Wireless M-Bus Virtual Water Meter button pressed function.

Parameters

N/A		

This function handles the button press event for the Wireless M-Bus virtual water meter sensor, in this case, it resets the total water consumption.

Returns

- sl_status_t: SL_STATUS_OK if button press was successful SL_STATUS_FAIL if button pressed failed

Definition at line 112 of file
/mnt/raid/workspaces/ws.RVnFspEgR/overlay/gsdk/app/rail/component/sl_rail_sdk_wmbus_sensor_virtual_water_meter/sl_rail_sdk_wmbus_sensor_virtual_water_meter.h

Configurations

Configurations

Macros

```
#define SL_RAIL_SDK_WMBUS_SENSOR_VIRTUAL_WATER_METER_SLEEPTIMER_PRIORITY 10
Default sleeptimer priority.

#define SL_RAIL_SDK_WMBUS_SENSOR_VIRTUAL_WATER_METER_SLEEPTIMER_TIMEOUT 1000
Default sleeptimer timeout.
```

Migration Guide

Migration Guide

RAIL SDK 2.19.0

Highlights:

- Uses RAIL LIB 2.19.0
- Examples now use `sl_main` instead of `sl_system`

Detailed API Change

- API Naming:
 - Switched from `CamelCase` to `snake_case` for most APIs and typedefs.
 - All public APIs now use `sl_rail_handle_t`, `sl_rail_status_t`, etc., instead of the old `RAIL_Handle_t`, `RAIL_Status_t`, etc.

Breaking Changes in Version 2.19.0

Example of API Change:

```
// Version 2.18.2 (OLD)
uint16_t unpack_packet(
    uint8_t *rx_destination,
    const RAIL_RxPacketInfo_t *packet_information,
    uint8_t **start_of_payload,
    phy_modulation_e modulation
);

// Version 2.19.0 (NEW)
uint16_t unpack_packet(
    sl_rail_handle_t rail_handle, // New first parameter
    uint8_t *rx_destination,
    const sl_rail_rx_packet_info_t *packet_information, // Renamed type
    uint8_t **start_of_payload,
    phy_modulation_e modulation
);
```

Manual Update for Range Test Projects:

- If your project directory name contains `range_test`, you should manually:
 - Remove any lines declaring `static RAIL_Handle_t rail_handle;` from all `.c` and `.h` files.
 - Remove `rail_handle` from function parameter lists, regardless of its position (beginning, middle, or end).
 - Remove any assignments to `rail_handle` and any standalone usages of `rail_handle` in your code, to ensure no obsolete references remain.

RAIL SDK 2.18.0

Directory and SDK Structure Changes

- SiSDK-2024.12 Release:
 - RAIL and Connect SDKs are now separated.

The former FLEX SDK is split into:

- RAIL SDK
- Connect SDK
- Directory changes: files moved from `app/flex` to `app/rail` and `app/connect`.

Component and API Renames

- Component file names and APIs updated:
 - Any occurrence of `flex` in APIs or definitions updated to match the new structure.

Table of Changes

Old File	New File	Changes	Details
<code>simple_rail_assistance.h</code>	<code>sl_rail_sdk_simple_assistance.h</code>	-	
<code>simple_rail_heartbeat.h</code>	Deleted	-	
<code>simple_rail_rx.h</code>	Deleted	-	
<code>simple_rail_rx_cli.h</code>	Deleted	-	
<code>simple_rail_tx.h</code>	Deleted	-	
<code>sl_flex_mode_switch.h</code>	<code>sl_rail_sdk_mode_switch.h</code>	Rename API: <code>prepare_package</code> -> <code>prepare_packet</code>	
<code>sl_flex_ieee802154_support.h</code>	<code>sl_rail_sdk_ieee802154_support.h</code>	Change part of names: FLEX -> RAIL_SDK , flex -> rail_sdk	
<code>sl_flex_rail_package_assistant.h</code>	sl_rail_sdk_packet_assistant.h	Rename API: <code>prepare_package</code> -> <code>prepare_packet</code>	
<code>sl_flex_packet_asm.h</code>	<code>sl_rail_sdk_packet_asm.h</code>	Change part of names: FLEX -> RAIL_SDK, flex -> rail_sdk	
<code>sl_flex_rail_sleep.h</code>	sl_rail_sdk_sleep.h	Rename API: <code>sl_flex_rail_sleep_init</code> -> <code>sl_rail_sdk_sleep_init</code>	
<code>sl_flex_util_802154_init.h</code>	<code>sl_rail_sdk_util_802154_init.h</code>	Change part of names: <code>sl_flex_util_init</code> -> <code>sl_rail_sdk_util_init</code> , <code>sl_flex_util_get_handle</code> -> <code>sl_rail_sdk_util_get_handle</code>	
<code>sl_flex_util_ble_init.h</code>	<code>sl_rail_sdk_util_ble_init.h</code>	Change part of names: <code>sl_flex_util_init</code> -> <code>sl_rail_sdk_util_init</code> , <code>sl_flex_util_get_handle</code> -> <code>sl_rail_sdk_util_get_handle</code>	
<code>sl_flex_util_ble_protocol.h</code>	<code>sl_rail_sdk_util_ble_protocol.h</code>	Rename include <code>sl_flex_util_ble_protocol_config</code> -> <code>sl_rail_sdk_util_ble_protocol_config</code>	
<code>sl_rail_simple_cpc.h</code>	<code>sl_rail_sdk_simple_cpc.h</code>	Change part of names: rail -> rail_sdk	
<code>sl_wmbus_packet_assembler.h</code>	sl_rail_sdk_wmbus_packet_assembler.h	Change part of names: wmbus -> rail_sdk_wmbus	
<code>sl_wmbus_sensor_pulse_counter.h</code>	sl_rail_sdk_wmbus_sensor_core.h	Change part of names: wmbus -> rail_sdk_wmbus	
<code>sl_wmbus_sensor_pulse_counter.h</code>	sl_rail_sdk_wmbus_sensor_pulse_counter.h	Change part of names: wmbus -> rail_sdk_wmbus	

Old File	New File	Changes	Details
sl_wmbus_sensor_thermometer.h	sl_rail_sdk_wmbus_sensor_thermometer.h	Change part of names: wmbus -> rail_sdk_wmbus	
sl_wmbus_sensor_virtual_water_meter.h	sl_rail_sdk_wmbus_sensor_virtual_water_meter.h	Change part of names: wmbus -> rail_sdk_wmbus	

Using RAIL for Wireless M-Bus Applications with EFR32

Using RAIL for Wireless M-Bus Applications with EFR32

This section replaces *AN1119: Using RAIL for Wireless M-Bus Applications with EFR32*. Further updates to this application note will be provided here.

This application note describes using the Flex SDK for Wireless M-Bus development on EFR32 Wireless Geckos. It includes features and limitations, using the radio configurator and supplied software components, and a description of the examples. Proprietary is supported on all EFR32FG devices. For others, check the device's data sheet under Ordering Information > Protocol Stack to see if Proprietary is supported. In Proprietary SDK version 2.7.n, Connect is not supported on EFR32xG22.

Overview

Wireless Gecko supports all Wireless M-Bus PHY configurations, according to EN13757-4 (both directions in all cases):

- Mode S
- Mode T
- Mode R2
- Mode C
- Mode N (all indexes, index 3/9/12 only on Series 2)
- Mode F

Note that, currently, only EFR32xG23 supports Wireless M-Bus on Series 2.

For the RF performance of the above modes, see [AN1076: EFR32FG1x Wireless M-Bus Performance Measurement Results](#).

In all modes, we support basic data link layer features for both frame format A and frame format B:

- Segmenting frames into blocks, and reassembling them to frame
- CRC calculation and checking
- Frame length decoding
- Data encoding and decoding (Manchester and 3 out of 6)
- Postamble transmission

The software is built on top of RAIL, therefore it inherits RAIL features that can be useful for implementing Wireless M-Bus, such as timestamping or scheduled reception and transmission.

The supplied example includes an easily reusable module that provides some useful features, including mode 5 (AES-128 with dynamic initialization vector) security.

Simplicity Studio 5, used with the Proprietary Flex SDK v3.x, introduced an updated radio configurator graphical user interface (GUI). The illustrations in [Using the Radio Configurator](#) have been updated. Places where the functionality diverges from that in Simplicity Studio 4, used with Flex SDK v2.x, are noted. This document applies to both variants.

This document is mainly about the examples provided through Simplicity Studio, which can be used as a guideline when developing a Wireless M-Bus stack or a very simple application. If you need a fully-featured Wireless M-Bus stack, it is commercially available from our 3rd party partner company. For more details, contact our sales team.

Limitations

Limitations

T Mode Meter to Other Device (only for EFR32 Series 1)

While the hardware supports 3 out of 6 coding, EFR32 Series 1 cannot generate the postamble required by this mode (this does not matter in receive mode). For T Mode Meter to Other transmission on EFR32 Series 1, we recommend using the supplied encoder function (`WMBUS_phy_software()`), which calculates the CRC using the GPCRC peripheral, encodes the packet using a software based 3 out of 6 encoder, and adds the required postamble.

On EFR32 series 2, where Wireless M-Bus support is available (e.g. EFR32xG23), this feature is available from hardware. The software workaround is no longer recommended.

Select Frame Type Based on Sync Word

Modes C, N, and F support frame formats A and B, where the sync word selects the frame format. Receiving frame A and frame B with the same configuration is currently not supported. You can either:

- Set up the frameA configuration and receive only frame format A.
- Set up the frameB configuration and receive only frame format B.
- Manually create a multi-PHY configuration and switch between the configurations at runtime. However, you must decide which frame you expect before you switch to RX mode.
- Set up a custom C/N/F configuration with no frame coding option. In that case, you must decode the length and check CRC in software, but you can decide the frame type during reception as well.

Modes that support frame format B always have both sync words configured:

- FrameA and noFrame configurations have sync word 0 configured for frame type A and sync word 1 for type B.
- FrameB configurations have sync word 0 configured for frame type B and sync word 1 for frame type A.

You cannot receive frame type B with a frameA configuration or vice versa, but you can enable the second sync word in RAIL, and use the sync detect event to recognize that you are receiving something with the other frame type.

Decoder for Both C and T Mode Meter to Other

It is theoretically possible to receive mode C frames with a mode T configuration, because the first 10 bits of the sync word are the same. If the next 6 bits are a valid 3 out of 6 code, it is a T frame, and if the next 16 bits are the second part of mode C's sync word, it is a C frame. Since the second part of mode C's sync word is not a valid 3 out of 6 code, it is not possible that both conditions are true for a correctly formatted frame.

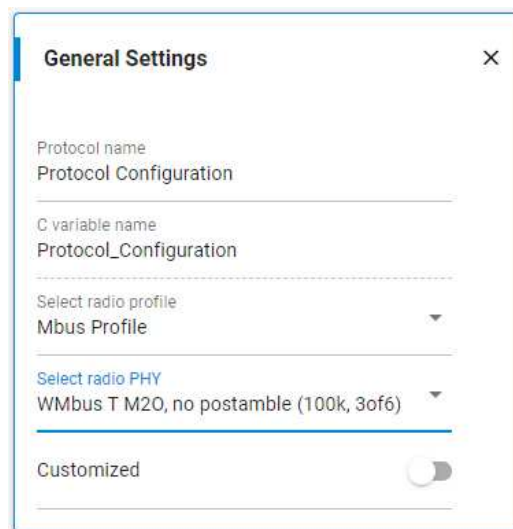
With the *WMBus TC M2O (100k, frameA)* PHY, it is possible to receive mode T and mode C packets. When transmitting, this PHY always uses mode T. The `RAIL_WMBUS_Config` API can be used to enable the simultaneous RX option which makes it possible to receive mode C frame A/B and mode T frame type A packets.

Note, that with the T+C mode PHY, performance cannot be guaranteed. The PHY is based on the mode T PHY, so it should match the mode T performance, but this specific mode was not characterized.

Using the Radio Configurator

Using the Radio Configurator

For Wireless M-Bus configurations, always use the Mbus Profile:



Multi-PHY Configuration

The following modes require multiple PHYs:

- Mode T2 (bidirectional only)
- Mode C2 (bidirectional only)
- Mode N has a separate PHY for each bitrate (index 1-5-7-10; 2-6-8-11; 4-18 and 3-9-13). In the 2013 release of EN13757-4, the naming was different. See the tables in Recommended Configurations below.
- Modes C, N, and F have separate PHYs for frame formats A and B

The Radio Configurator can be used to set up virtual channels. For example, channel 0 can be used for *Mode T meter to other* and channel 1 can be used for *Mode T other to meter*.

There are other possible uses, such as supporting all 868MHz based modes (S, T and C) with simple channel changes.

It is also possible to configure an application with multiple protocols as Wireless M-Bus modes. In order to change protocols RAIL_ConfigChannels must be used. This can be useful to select a mode during boot or configuration. For further details, if you are working with Proprietary Flex SDK v2.x see [AN971: EFR32 Radio Configurator Guide for RAIL in Simplicity Studio v4](#). If you are working with Flex SDK v3.x see [AN1253: EFR32 Radio Configurator Guide for Simplicity Studio 5](#).

The **Wireless M-bus Meter** example application is configured to support Mode T2 bidirectional mode using Multi-PHY configuration.

Recommended Configurations

For 868 MHz and 434 MHz bands

Mode	Submode	Single PHY Configuration Name (frame A) (1)	Single PHY Configuration Name (frame B) (1)	Freq. (MHz)
Mode S	N/A	WMbus S (32.768k, Manchester)		868.3
Mode T	Meter to Other, Rx	WMbus T M2O (100k, 3of6) (2) WMbus T M2O, no postamble (100k, 3of6) (3)		868.95
	Meter to Other, Tx	WMbus T M2O (100k, 3of6) (2) WMbus T M2O (100k, no framing) (4)		868.95
	Other to Meter	WMbus T O2M (32.768k, Manchester)		868.3
Mode TC	Meter to Other	WMbus TC M2O (100k, frameA) (2) (5)		868.95
Mode R2	N/A	WMbus R2 (4.8k, Manchester)		868.33
Mode C	Meter to Other	WMbus C M2O frameA (100k) (6)	WMbus C M2O frameB (100k) (7)	868.95
	Other to Meter	WMbus C O2M frameA (50k) (6)	WMbus C O2M frameB (50k) (7)	869.525
Mode F	N/A	WMbus F, frameA (2.4k) (6)	WMbus F, frameB (2.4k) (7)	433.82

Notes:

1. The names were changed from earlier releases to make them more uniform, but they always included the mode, the bitrate, and the frame type (where applicable)
2. Only for EFR32 series 2 with Wireless M-Bus support.
3. Only for EFR32 series 1. Could also work in Tx mode, but it does not generate the postamble required by the standard.
4. Only for EFR32 series 1. Should be used with the supplied software encoder (`WMBUS_phy_software()` function).
5. See [Decoder for Both C and T Mode Meter to Other](#) for details.
6. Sync word for frame format A is set up for sync word 0, sync word for frame format B is set up as sync word 1 but is not enabled by default. Receiving frame format B is not possible, but the sync detect event could be used.
7. Sync word for frame format B is set up for sync word 0, sync word for frame format A is set up as sync word 1 but is not enabled by default. Receiving frame format A is not possible, but the sync detect event could be used.

For the 169 MHz band (Mode N)

Index (according to EN13757-4-2019)	Single PHY configuration (Frame A) (1,2)	Single PHY configuration (Frame B) (1,3)	Freq. (MHz)	Channel Spacing (kHz)	Last channel number
1 (4)	WMbus N, index 1/5/7/10, frameA (2.4k)	WMbus N, index 1/5/7/10, frameB (2.4k)	169.40625	12.5	5
2 (5)	WMbus N, index 2/6/8/11, frameA (4.8k)	WMbus N, index 2/6/8/11, frameB (4.8k)	169.40625	12.5	5
3 (7)	WMbus N, index 3/9/12, frameA (6.4k)	WMbus N, index 3/9/12, frameB (6.4k)	169.41	12.5	5
4 (6)	WMbus N, index 4/13, frameA (19.2k)	WMbus N, index 4/13, frameB (19.2k)	169.4375	50	0
5	WMbus N, index 1/5/7/10, frameA (2.4k)	WMbus N, index 1/5/7/10, frameB (2.4k)	169.48125	12.5	0
6	WMbus N, index 2/6/8/11, frameA (4.8k)	WMbus N, index 2/6/8/11, frameB (4.8k)	169.48125	12.5	0

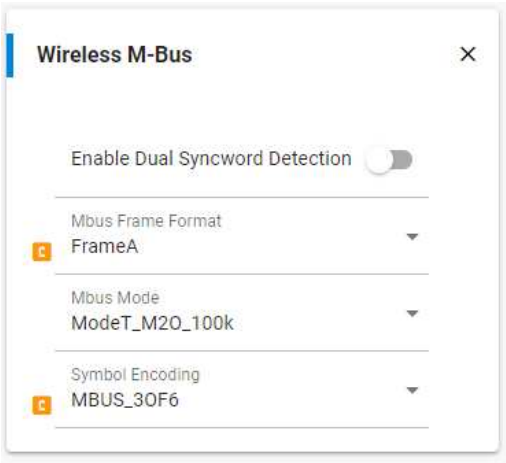
Index (according to EN13757- 4-2019)	Single PHY configuration (Frame A) (1,2)	Single PHY configuration (Frame B) (1,3)	Freq. (MHz)	Channel Spacing (kHz)	Last channel number
7	WMbus N, index 1/5/7/10, frameA (2.4k)	WMbus N, index 1/5/7/10, frameB (2.4k)	169.49375	12.5	7
8	WMbus N, index 2/6/8/11, frameA (4.8k)	WMbus N, index 2/6/8/11, frameB (4.8k)	169.49375	12.5	7
9	WMbus N, index 3/9/12, frameA (6.4k)	WMbus N, index 3/9/12, frameB (6.4k)	169.41	12.5	57
10	WMbus N, index 1/5/7/10, frameA (2.4k)	WMbus N, index 1/5/7/10, frameB (2.4k)	169.59375	12.5	17
11	WMbus N, index 2/6/8/11, frameA (4.8k)	WMbus N, index 2/6/8/11, frameB (4.8k)	169.59375	12.5	17
13	WMbus N, index 4/13, frameA (19.2k)	WMbus N, index 4/13, frameB (19.2k)	169.625	50	3

Notes:

- 1. The names were changed from earlier releases to make them more uniform, but they always included the mode, the bitrate, and the frame type (where applicable)
- 2. Sync word for frame format A is set up for sync word 0, sync word for frame format B is set up as sync word 1 but is not enabled by default. Receiving frame format B is not possible, but the sync detect event could be used.
- 3. Sync word for frame format B is set up for sync word 0, sync word for frame format A is set up as sync word 1 but is not enabled by default. Receiving frame format A is not possible, but the sync detect event could be used.
- 4. Submodes, according to EN13757-4-2013, available on this index: N1c, N2c (on ch1); N1d, N2d (on ch3).
- 5. Submodes, according to EN13757-4-2013, available on this index: N1a, N2a (on ch0); N1b, N2b (on ch1); N1e, N2e (on ch4); N1f, N2f (on ch5).
- 6. Same as submode N2g in EN13757-4-2013.
- 7. Only available on EFR32 Series 2 with Wireless M-Bus support.

Using Custom Settings

All the above configurations (and more) can be set up using Custom settings. Currently, to use the multi-PHY features, you have to use custom settings for all but the first (virtual) channel.

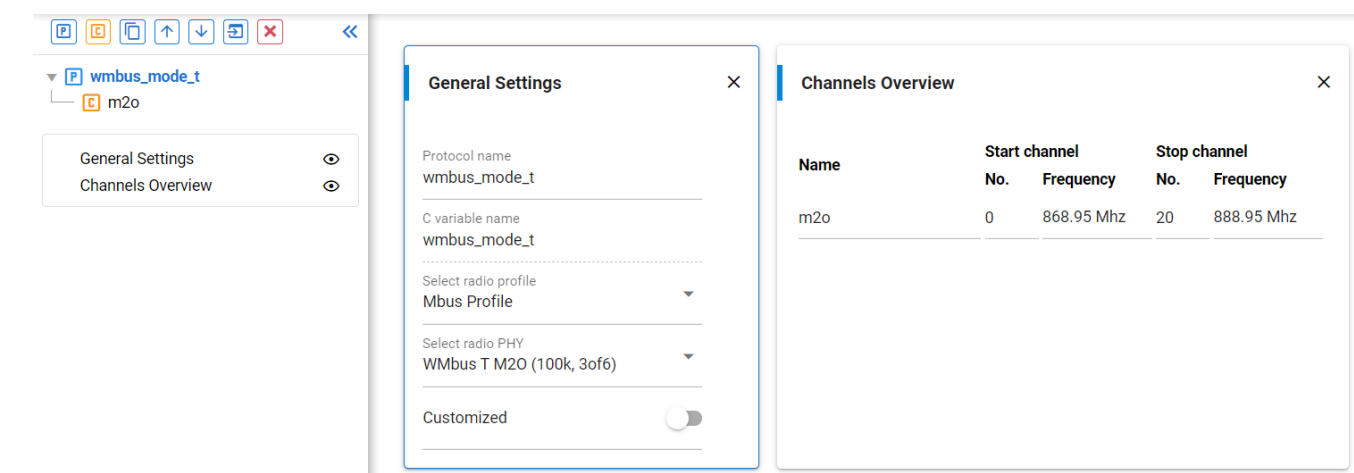


For details about the Wireless M-Bus custom setting parameters, see [AN1253: EFR32 Radio Configurator Guide for Simplicity Studio 5](#).

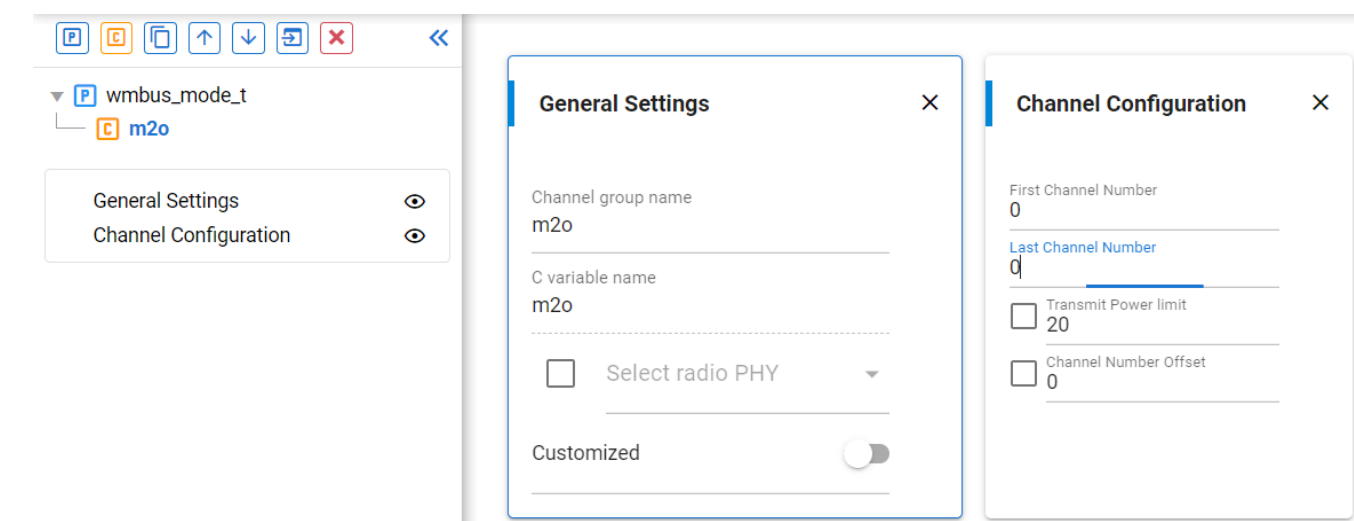
Using Multi-PHY Features

The following procedure shows how to set up a T2 meter/collector on Series 2 as an example, using Simplicity Studio 5 and Flex SDK v3.x. The settings to use are the same as in earlier configurator versions, but the Multi-PHY configurator workflow changed significantly between Simplicity Studio 4 and Simplicity Studio 5. To apply these changes in Simplicity Studio 4, see [AN971: EFR32 Radio Configurator Guide for RAIL in Simplicity Studio v4](#).

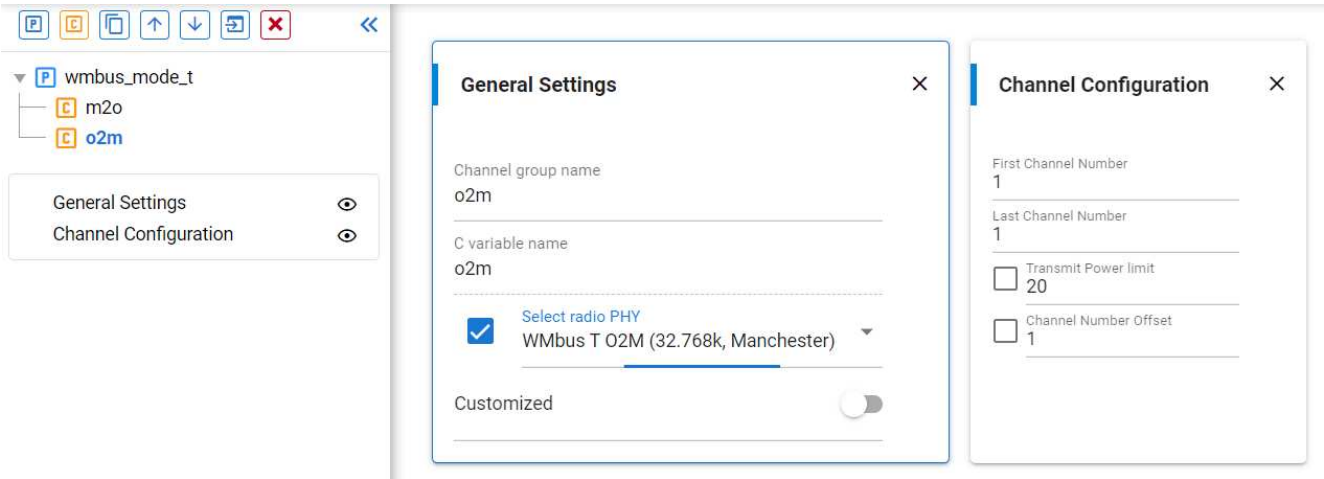
First, select the PHY you wish to use as channel 0 under the protocol setup. In the example, this would be M2O TRx setup, which is *WMbus T M2O (100k, 3 of 6)*.



With this, the default channel group is already set for Mode T M2O. All that remains is to set the channel numbering and the name.



Create a new channel group by clicking the [C] control on the menu in the upper left and select the O2M TRx setup, which is *WMbus T O2M (32.768k, Manchester)*. Set the channel numbering so that channel 1 will correspond to this PHY.



With this setup, a meter can transmit on channel 0 and receive on channel 1, while a collector can transmit on channel 1 and receive on channel 0.

Wireless M-Bus Support Component

The Wireless M-Bus Support Component

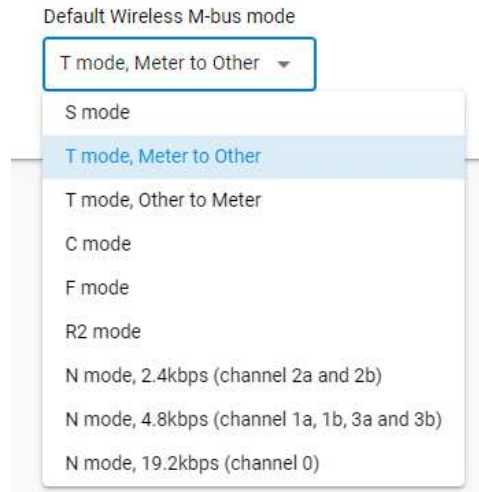
Note: In Flex SDK 3.x, the component can be found under Flex/RAIL/Utility. In Flex SDK v2.x, the wmbus plugin was equivalent to this component.

This component provides the following features:

- Processes the packet before sending it, if needed. Currently needed for mode T meter to other only.
- Helper functions to get the timing in limited access mode.
- Helper functions to fill the payload with EN13757-3 compatible payload.
- Helper functions to encode/decode manufacturer field.
- Helper functions for cryptography using mbedTLS. Currently only supports mode5 (AES128-CBC with dynamic init vector).
- Helper types for data link and transport layer header.

For detailed documentation, see `sl_wmbus_support.h` (or in Flex v2.x `wmbus.h`), documented via Doxygen style.

In Flex 3.x, the component allows configuration of the wmbus mode used. This selects the timing provided to the application in limited access mode. If *T mode, Meter to Other* is selected and the part requires it, it also enables the workaround described in [Limitations](#).



Example Application

Example Application

The provided example application pair implements a very basic meter-collector interaction.

Meter

Note: In Flex SDK 2.x, both `TX_CHANNEL` and the mode variable can be found in `main.c`.

The meter periodically sends synchronous SND-NR messages with some hardcoded value with mode 5 security. While waiting, it sleeps in EM2 in idle or EM1 if the main oscillator is required for scheduling or Rx mode. By default, the meter is configured for limited access mode (short receive window after transmission), but it doesn't handle any received packets, it is just implemented to demonstrate the scheduling required for an application.

The application will always transmit on the channel defined by `TX_CHANNEL` in `app_init.h` (0 by default). In symmetric modes (S, R2, N, F), `TX_CHANNEL` is also used for reception. In asymmetric modes (T and C), `TX_CHANNEL+1` is used for reception.

In Flex SDK 3.x, be sure to select the correct mode in the wireless M-bus support component configuration, as described in [The Wireless M-Bus Support Component](#). In Flex SDK 2.x, the same configuration should be applied in the `main.c` file:

```
static const WMBUS_Mode_t mode = WMBUS_MODE_T_METER
```

The meter application also includes `wmbus_sample_frame.c` and `.h`, which can be used for guidance on how to assemble a simple wmbus frame.

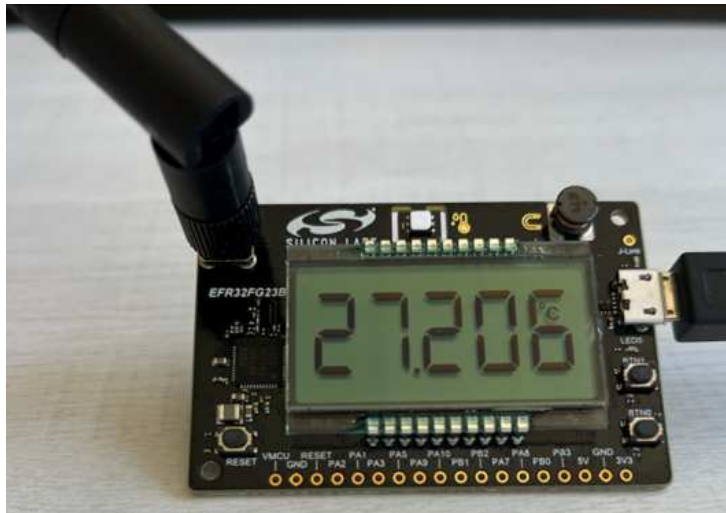
Meter Peripherals

The meter example, depending on the devkit it is running on, implements various sensors:

- Virtual water meter sensor, which periodically increments.
- Pulse counter, which increments on button press of BTNO. In the M-Bus frame, this is transmitted as energy in watt-hours (Wh).
- Thermometer, which uses the on-board Si7021 to get temperature measurements.

The latter two are only available on development kits which have the required peripheral.

On FG23-DK2600, the 5-digit 7-segment LCD shows the measurement of one of the sensors. The active sensor can be selected by pressing BTN1.



The meter device enters EM2 sleep when in idle. It provides a command line interface on serial port at 9600 baud, to allow it to work in EM2 sleep. On the command line interface, the active sensor can be selected.

Collector

The collector prints the received packet on serial terminal, with some information (like the first block) detailed. If the packet is EN13757-3 compatible with the short header, it also decodes mode5 security, if required.

Once the meter and collector are running, the collector should print messages like this to the serial terminal:

```
RX:[Time:163580960]
Block-1:[L:30,C:0x44,M:SIL,ID:00000001,Version:0x01,devType:0x07]
AppHeader:[Cl:0x7A,AccessNr:60,Status:0x00,encMode:5,Accessibility:02,encBlocks:1,sync:1]
[0x2F 0x2F 0x04 0x13 0x39 0x30 0x00 0x00 | 0x02 0x3B 0x7B 0x00 0x2F 0x2F 0x2F 0x2F]
```

A Wireless M-Bus sniffer can also be used. In that case, the default crypto key used by the application is:

```
00112233445566778899AABBCCDDEEFF.
```

For unencrypted packets, RAILTest also can be used as a sniffer with the right meter to another device to receive configuration.

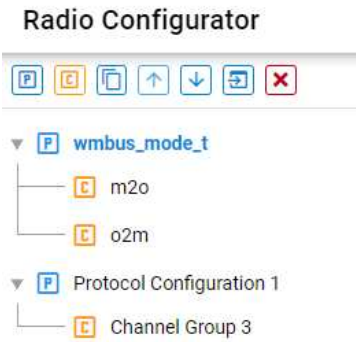
Changing the Mode of the Example

Changing the Mode of the Example

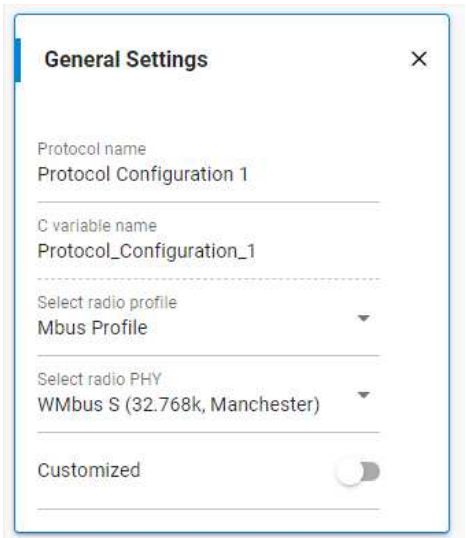
This section provides summary instructions on changing the mode of the meter example. Full details on the Radio Configurator workflow are provided in [AN1253: EFR32 Radio Configurator Guide for Simplicity Studio 5](#).

Changing to a Symmetric Configuration

Because the default modeT configuration requires a multi-PHY setup, you would need to remove all the channel overrides to completely undo it. A simpler way is to add a new protocol with the "P" control in the Radio Configurator, and remove the old one later.



On the new protocol, select the Mbus Profile, and the PHY you want, for example *WMBus S (32.768k, Manchester)*.



Save the config, which will generate its files. Next, open the configuration of the *Wireless M-bus support* component.

rail_soc_wmbus_collector

OVERVIEWSOFTWARE COMPONENTSCONFIGURATION TOOLS

Filter components by

Configurable

Installed

Installed by you

SDK Extensions

Quality

Search keywords, component's name

wmbus

Flex

RAIL

Utility

Wireless M-bus support

Platform

Radio Configuration

Wireless M-bus support

Configure

Description

Adds minimalist Wireless M-Bus support for coding and decoding packets.

Quality

PRODUCTION

Set the mode to match the Radio Configurator.

Available Wireless M-bus mode options for Wireless M-bus Meter app

Default Wireless M-bus mode

S mode

Remove the old protocol with the "X" control.

This changes the timing of the application according to EN13757-4 and, for EFR32 Series 1, removes the software coder needed for mode T. When a symmetric mode is selected, it also sets up the software to use the same channel for Tx and Rx.

Changing to an Asymmetric Configuration

For an asymmetric configuration, follow the same steps as above to configure the Tx PHY. For example, for modeC, set the Radio Configurator for *Wmbus C M20 frameA (100k)* and set the component configuration to *C mode*.

However, you must also set up an Rx configuration for channel 1. First, open the Radio Configurator again, and set the *stop channel No.* of the Tx PHY to 0.

Radio Configurator

SearchView

Protocol Configuration 1

Channel Group 3

General Settings

Channels Overview

Select radio PHY

Wmbus C M20 frameA (100k)

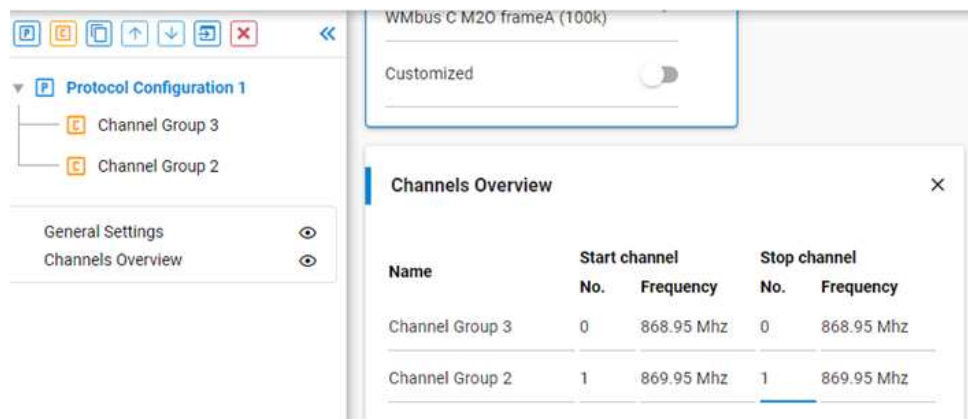
Customized

Channels Overview

Name	Start channel No.	Frequency	Stop channel No.	Frequency
Channel Group 3	0	868.95 Mhz	0	868.95 Mhz

The original setting was 20, but in a symmetric config this does not matter because only channel 0 is used, when the mode in the component is configured correctly.

Next, click the "C" control to add a new Channel Group, and set both its start and stop channel to 1 (although, technically, the latter is not needed).

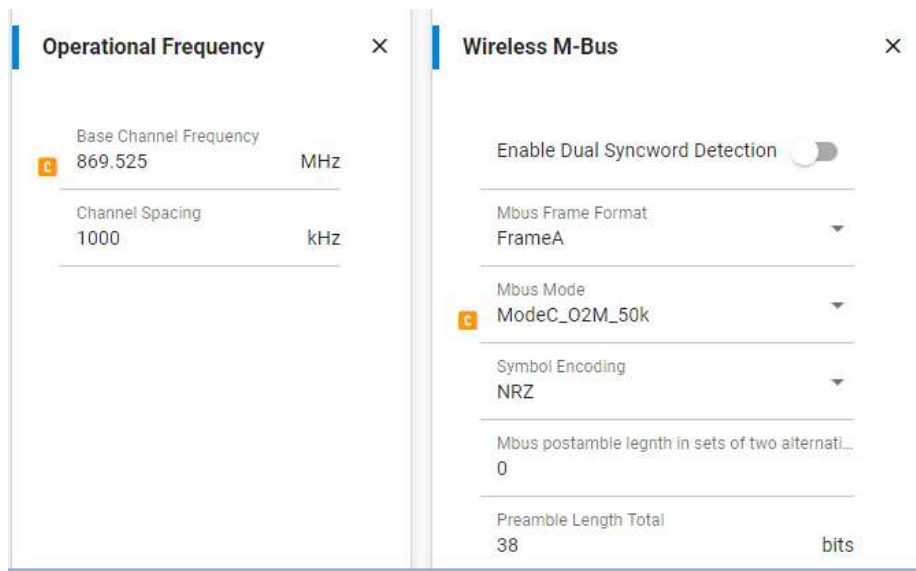


Select the new channel group and enable customization. Because the radio configurator only supports a single preconfigured PHY in channel-based Multi-PHY, which is the most effective method for this setup, you must manually configure this PHY for modeC O2M.

Finally, set up the following PHY parameters, according to EN13757-4:

- Base Frequency
- Mbus Frame Format
- Mbus Mode
- Symbol Encoding
- Preamble Length Total
- Mbus Postamble Length

You can also create a new protocol set up to the O2M config and copy the settings from there. For example, the following figure shows the settings for ModeC.



Save the changes in the Radio Configurator. Since the mode is already set up in the component, the configuration change is complete, and you can compile and use the project.

Overview

RAIL Training Overview

The training section provides a series of documentation that helps developers learn the basics of RAIL.

The contents here are intended for embedded C developers who are already somewhat familiar with the platform and development tools, e.g. the [Getting Started](#) section.

While the section is written for firmware developers, as proprietary wireless often requires knowledge on the physical layer, some knowledge on the topic is required, or at least highly recommended. The radio configurator, where the physical layer can be configured is documented in [AN1253](#)

Note that the pages here are not automatically updated with every SDK release. If there's a conflict between these articles and the [RAIL API documentation](#), always trust the API documentation. If you find an issue, please let us know on our [community](#).

Getting Started with RAIL

You should start with these tutorials. All others are based on these, as these are the minimum to develop useful applications on RAIL.

- [Introduction, RAIL components and the Empty Example](#)
- [Transmitting a Packet](#)
- [Event Handling and Automatic State Transitions](#)
- [Downloading received messages](#)
- [Time, Timestamping and Scheduling](#)

Basics

These articles are common and basic features. While technically not required for basic functionality, it's highly recommended to read them before developing any production quality project.

- [Debugging](#): Debug features of RAIL, and recommended methods of debugging RAIL applications
- [Calibrations](#): Calibrations controllable from RAIL
- [Power Manager integration](#): How RAIL Integrates with Power Manager
- [Idle Modes](#): The explanation of various parameters of `RAIL_Idle()` .

Further Topics

These articles document some specific features of RAIL that some applications need. It is recommended to read them if you think you will need that feature.

- [Timer Synchronization and Sleep](#): How RAIL can keep time in EM2 with the help of the RTCC
- [Tx/Rx Options](#) - The various options available for Rx and Tx. Mainly, Listen Before Talk and Address Filtering
- [Low duty cycle mode](#): The theory of duty cycling, a way to save energy on the Rx side by sacrificing some on the Tx side.
- [Interrupt and Thread Safety](#): While RAIL is, in general not thread safe, you actually only need to avoid specific operations, which are detailed here.
- [Understanding RAIL Config Files](#): Basic explanation of the various variables in `rail_config.c`

Multiprotocol

Multiprotocol is a complex topic, and it is recommended only if you have a good understanding of RAIL's basic functionality. This section describes both dynamic multiprotocol capabilities and MultiPHY capabilities.

- [Introduction](#): Explanation of Multiprotocol and MultiPHY, and recommendation what to use for given problem.
- [MultiPHY Usecases](#): Explanation of MultiPHY through the most typical usecases
- [DMP](#): A short introduction of Dynamic Multiprotocol, highlighting the differences between Single protocol and Multiprotocol usage of RAIL.

Introduction

Introduction, RAIL components, and the Empty Example

What is RAIL?

RAIL is short for Radio Abstraction Interface Layer, and is the most direct interface available for EFR32 radios. You can think of it as a radio driver. While this sounds like a restriction, this actually makes software development for proprietary wireless much simpler:

- EFR32 has a very flexible radio, which also makes it quite complex. RAIL (and the radio configurator) provides an easy-to-use interface of EFR32's features.
- The various generations of EFR32s are slightly different, but the RAIL API is the same (except maybe the chip specific updates), which makes hardware updates almost seamless.
- We plan to add RAIL support to all of our future wireless MCUs.

RAIL application development is currently possible in the SiSDK in Simplicity Studio. To progress further with this tutorial, please **remember to open the [RAIL API documentation](#)**.

When to Use RAIL?

If you have an existing protocol, and you must be compatible with it, you'd probably have to use RAIL (as all of our stacks use a standardized frame format).

If you want to use sub-GHz communication, you have a few options, including RAIL. For simple point to point communication, RAIL might be the simplest solution. However, as soon as you need security or addressing, it might be better choose a protocol stack, such as [Connect](#).

RAIL also has the benefit that it adds the least amount of delays to all radio events. In fact, some event notification will happen in interrupt context.

What is Covered by RAIL?

RAIL only supports what the radio hardware supports. For example, auto ACK and address filter is available, as the hardware provides support for it. On the other hand, while security is important for wireless communication, it's not really a radio task, hence RAIL does not support it. The crypto engine is an independent peripheral that can be used to encrypt payloads before giving them to RAIL and the radio hardware.

Supported Energy Modes

EM1p or higher is required for the radio to be operational (i.e., Transmitting, receiving, or waiting for packets, RAIL timer running from HF clock). On devices without EM1p support, EM1 is required for the radio. See [EFR32 Migration Guide for Proprietary Applications](#) for more details on EM1p.

EM2 or higher is required for RAIL scheduling or timers. In this mode, timers are running from LF clock, which needs configuration. This is configured by default in our examples, and will be discussed [in a later article](#) in detail.

EM3 or higher is required for RAIL to work without re-initialization.

Writing Code from Scratch

We do not recommend writing code from scratch because it is much simpler to have something existing that sets up the include paths and linker settings correctly. The best way to start is the example *RAIL - SoC Empty*.

RAIL Components

The empty application includes the following RAIL related components (under 'Platform/Radio'):

- *RAIL Library, Single Protocol*, which is the library itself
- *RAIL Utility, PA*, which configures and initializes the power amplifier
- *RAIL Utility, Initialization*, which initializes RAIL and loads an initial configuration
- *RAIL Utility, Callbacks*, which currently only implements the assertion callback
- *RAIL Utility, PTI*, which enables Packet Trace Interface, the hardware that feeds data to the Network Analyzer
- *RAIL Utility, RSSI*, which compensates the RSSI offset error (and can be fine tuned)

And a few other RAIL components, that are useful if you plan to use a RAIL-supported protocol, like IEEE 802.15.4.

There are a few components that you might want to install:

- *RAIL Utility, DMA*, which allocates a DMA channel to speed up RAIL initialization
- *Radio Utility, Front End Module*, which can be used to drive a FEM (or external PA/LNA), if your HW uses one
- *RAIL Utility, RF Path*, which can be used to select the antenna path on the parts that have multiple (like EFR32xG23)

If you work on a starter kit, you don't need to change the configuration of most of these components, except the Initialization component. However, the initialization component also includes a good starting point, that we don't need to modify for simple applications.

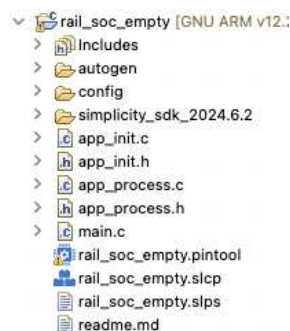
The Radio Configurator

The Radio Configurator is accessible by configuring the component *Advanced Configurators/Radio Configurator*. The usage of the radio configurator is out of scope for this tutorial series. For more details, see [AN1253: EFR32 Radio Configurator Guide](#).

The Project Structure

Projects always include the following parts:

- **autogen** folder: Only the autogen folder includes generated code. It includes the PHY configuration (`rail_config.c`), init code, the linker script, and other generated code used by components, like the command descriptors for the CLI interface
- **config** folder: Component configuration headers are placed into this folder. These can be edited with the Component Editor that can be opened on the Project Configurator via the Configure button, but directly editing the header file is also possible. The input of the radio configuration is also here, in `rail/radio_settings.radioconf`
- **simplicity_sdk** folder (with version number): Source and binary files added by components
- **files in the root folder**: Only the application specific files should be in the root folder, including source files, the project configurator (`.slcp` file) the Pin tool (`.pintool` file) and a readme file.



Note that all files related to the project should be visible in the project explorer, including header and library files.

Where to Add User Code

All projects include `main.c`, which is not recommended to modify. Instead, add the initialization code to `app_init.c`, and implement the main loop in `app_process.c`. This way, the *System* components can initialize components, and call the "process" function of the components that requires this. Additionally, enabling an RTOS will convert `app_process`'s main loop into an RTOS task.

Conclusion

This project is ready to send out frames. We're going to do that in the [next part](#).

Transmit

Transmitting a Packet

This tutorial is the continuation of a series started with [Introduction](#).

Please read it first if you haven't.

The completed example for this article is available on [GitHub](#). You can either import this example, as documented in the readme on Github, or follow this article and modify a **RAIL - SoC Empty** example. Note that the code snippets here just like the code on Github is missing a lot of error handling for easier readability. Therefore, it shouldn't be used as a base of development.

In this tutorial, we're going to modify the project *Flex (RAIL) - Empty Example* to transmit a packet. We're going to use the default, fixed length 16 Bytes payload configuration first.

This tutorial references some more advanced articles - we don't recommend reading them yet if you just started learning RAIL. They are only linked to highlight the connection.

Code snippets in this example are for illustration only. Refer to the attached files for reference code.

Buffer Handling

Setting Up the Packet Buffer

RAIL requires a buffer (usually called FIFO) to be configured for Tx. First, we have to allocate some memory for that buffer:

```
#define SL_TUTORIAL_TRANSMIT_BUFFER_LENGTH    256
SL_ALIGN(RAIL_FIFO_ALIGNMENT) static uint8_t tx_buffer[SL_TUTORIAL_TRANSMIT_BUFFER_LENGTH] SL_ATTRIBUTE_ALIGN(RAIL_FIFO_ALIGNMENT);
```

Note that you can't use any arbitrary size; EFR32 creates a FIFO ring buffer on this memory area, and it can only handle buffer sizes of power of 2 between 64 and 4096 (i.e., 64, 128, 256, 512, 1024, 2048, or 4096).

On all Series 2 chips, except EFR32xG21, RAIL requires a 4 bytes aligned FIFO. In the code above, we used `RAIL_FIFO_ALIGNMENT` which guarantees good alignment on any EFR32.

The Tx buffer can hold more than one packet, which can be useful if you need to send out a lot of messages quickly. To load the payload of the packet to the buffer, we have two options: write it via RAIL APIs or write to the memory directly.

Writing to the Buffer Directly

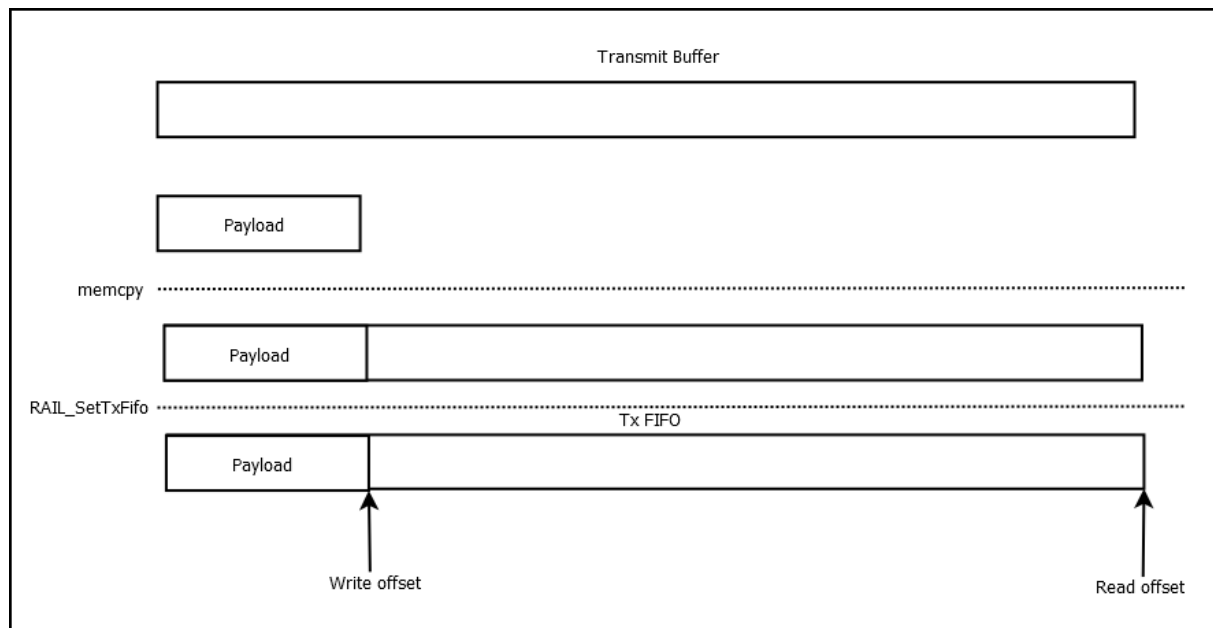
In `app_process.c`:

```
#define PAYLOAD_LENGTH 16
#define BUFFER_LENGTH 256
static const uint8_t payload[PAYLOAD_LENGTH] =
    {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
     0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

SL_ALIGN(RAIL_FIFO_ALIGNMENT) static uint8_t tx_buffer[BUFFER_LENGTH] SL_ATTRIBUTE_ALIGN(RAIL_FIFO_ALIGNMENT);

void load_payload_directly(RAIL_Handle_t rail_handle) {
    memcpy(tx_buffer, payload, PAYLOAD_LENGTH);
    RAIL_SetTxFifo(rail_handle, tx_buffer, PAYLOAD_LENGTH, BUFFER_LENGTH);
}
```

The `memcpy()` is just a standard C function to write to buffers, and we use `RAIL_SetTxFifo()` to pass that buffer to RAIL. We also tell RAIL how long the buffer is, and how much data it has already in it.



Note : This illustrates the simple case of one `RAIL_SetTxFifo()` call. In effect, different buffers are used for Rx and Tx.

Writing to the Buffer Indirectly

In `app_init.c` :

```
#define SL_TUTORIAL_TRANSMIT_BUFFER_LENGTH 256
SL_ALIGN(RAIL_FIFO_ALIGNMENT) static uint8_t tx_buffer[SL_TUTORIAL_TRANSMIT_BUFFER_LENGTH] SL_ATTRIBUTE_ALIGN(RAIL_FIFO_ALIGNMENT);

RAIL_Handle_t app_init(void)
{
    // Get RAIL handle, used later by the application
    RAIL_Handle_t rail_handle = sL_rail_util_get_handle(SL_RAIL_UTIL_HANDLE_INST0);

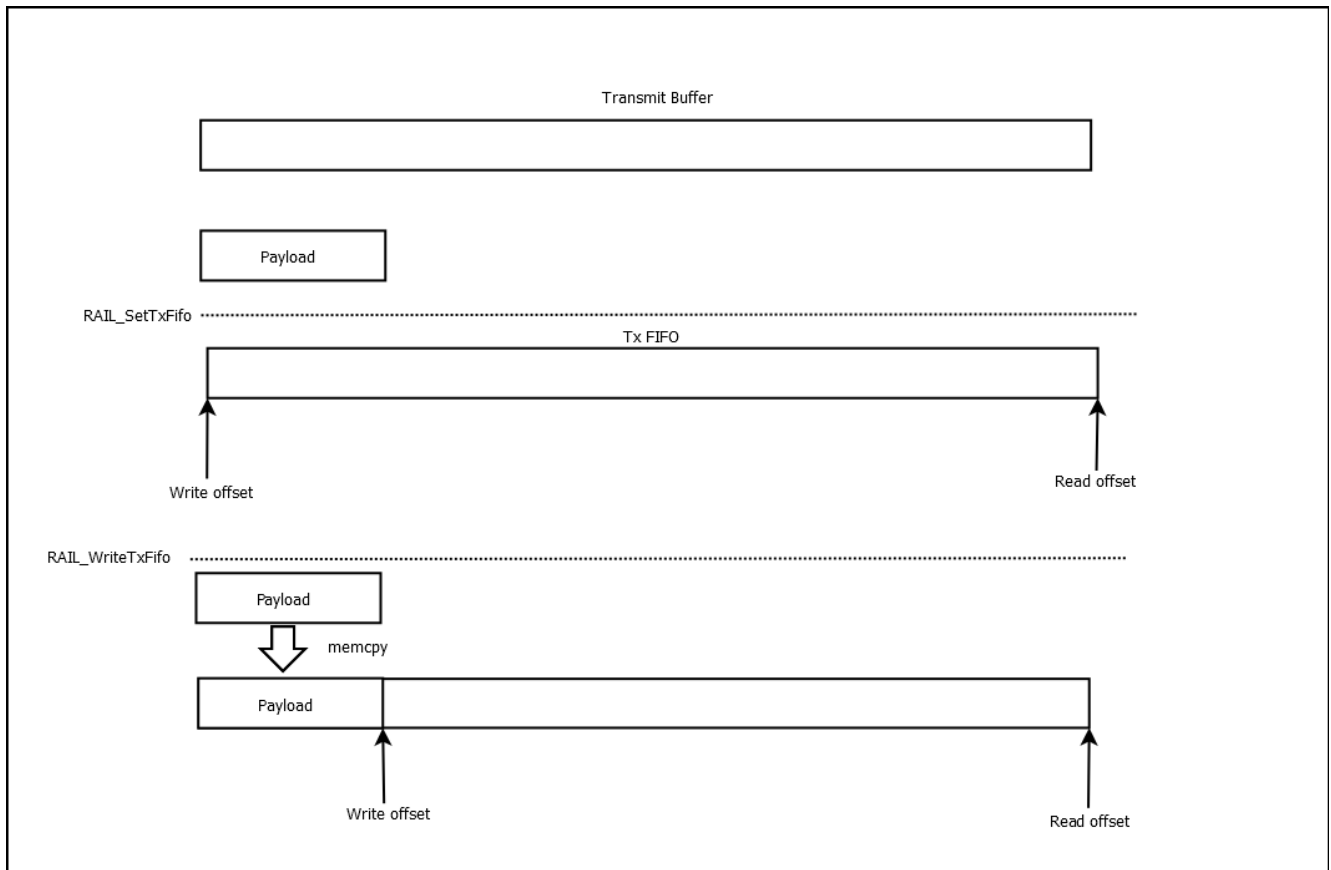
    RAIL_SetTxFifo(rail_handle, tx_buffer, 0, BUFFER_LENGTH);
    return rail_handle;
}
```

In `app_process.c` :

```
#define PAYLOAD_LENGTH 16
static const uint8_t payload[PAYLOAD_LENGTH] =
{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

static void loadPayloadIndirectly(RAIL_Handle_t rail_handle) {
  RAIL_WriteTxFifo(rail_handle, payload, PAYLOAD_LENGTH, false);
}
```

In this case, we pass the buffer to RAIL when it's still empty, and we write to that buffer using a RAIL API. Note that using `memcpy()` instead of `WriteTxFifo` would not work: while it would write the memory itself, it wouldn't change the read and write pointers of the FIFO, handled by RAIL, so RAIL would still think the buffer is empty.



Direct or Indirect

There are four main factors that could decide which method to use to write the FIFO:

- With indirect mode, the buffer can be used for partial packets. It is simpler to use for multiple packets. Buffer wrapping is also supported internally by RAIL.
- Calling `RAIL_SetTxFifo()` does not move any memory, just sets a few registers, while calling `RAIL_WriteTxFifo()` does need some time to copy the payload to the buffer.
- Calling `RAIL_SetTxFifo()` is not allowed during transmission. Whereas using `RAIL_WriteTxFifo()` is safe to use, even when radio transmission is occurring.
- Indirect mode clearly separates init and process code.

For simpler applications, `RAIL_SetTxFifo()` can be a better option. If you want to send almost the same packet, with just a few bytes modification, it's much better, since you don't have to write the whole packet again. (Note that if you want to transmit the *exact* same message over and over, you should use `RAIL_TX_OPTION_RESEND`, detailed in a [later article](#)).

On the other hand, the separation in RAIL examples between `app_init` and `app_process` works nicely with the indirect method, and in most real cases, you need to move memory anyway. If you want to send longer packets than your buffer, you must use `RAIL_WriteTxFifo()`, as you can call it during transmit. It's also useful if you want to send out a lot of packets: Use `RAIL_WriteTxFifo()` to load each message one after the other, then send them out quickly without any buffer operation.

The attached code uses indirect mode mainly for the clear separation.

FIFO Reset

The last parameter of `RAIL_WriteTxFifo` can reset the FIFO, which means it will invalidate the data already in there. This is also possible with `RAIL_ResetFifo`.

It's a good idea to run some tests without resetting the FIFO, as it can mask small errors, e.g., writing more to the FIFO than needed. However, in production, it's a good idea to reset the FIFO if you think it should be empty, as in production, masking errors are beneficial.

Transmitting the Packet

Starting the transmission is simple:

```
RAIL_StartTx(rail_handle, 0, RAIL_TX_OPTIONS_DEFAULT, NULL);
```

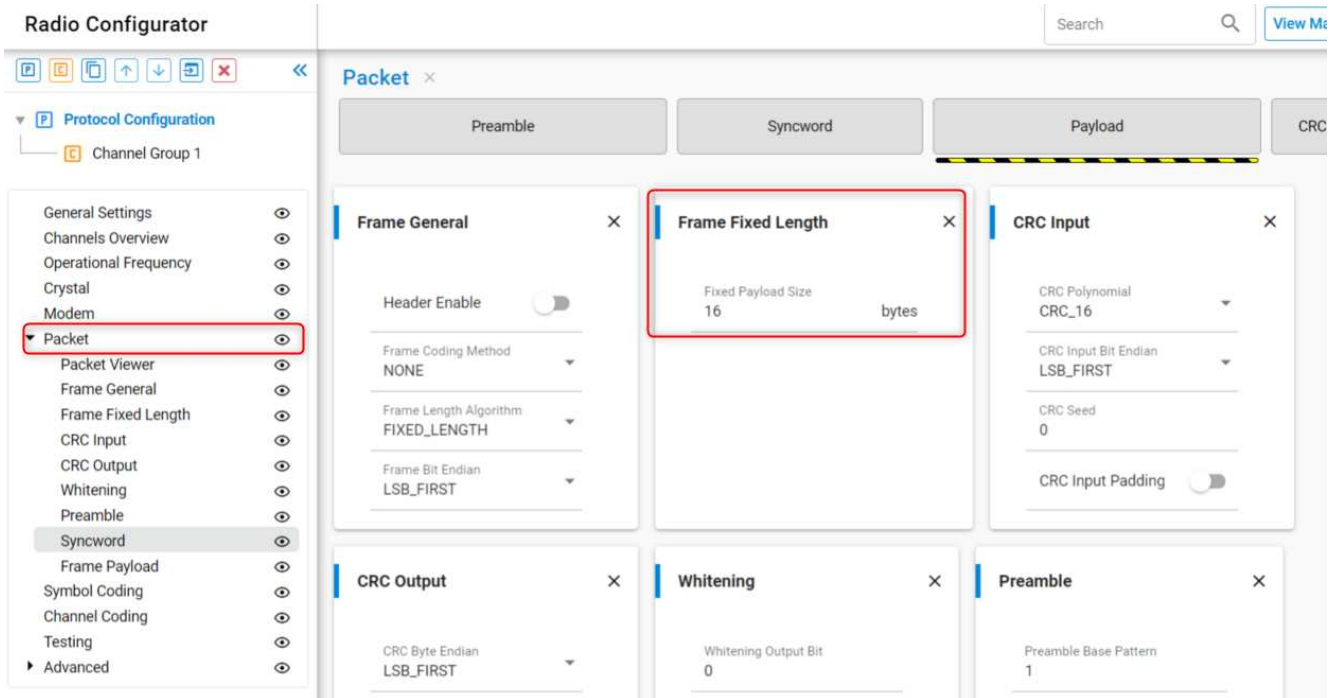
This instructs RAIL to start transmitting the data stored in the FIFO on channel 0 (second parameter). The third parameter can change various options, like using the second configured sync word (see an [article on the topic](#)). The last parameter is only required for [DMP \(dynamic multiprotocol\) mode](#).

Changing the Packet Length

You can change the configured fixed length on the Radio Configurator, but that's obviously not possible at runtime, which is often needed. **Note that the amount of data loaded into the FIFO does not matter as long as it's equal to or greater than the length of the frame.**

Changing Packet Length with a Fixed Length Configuration

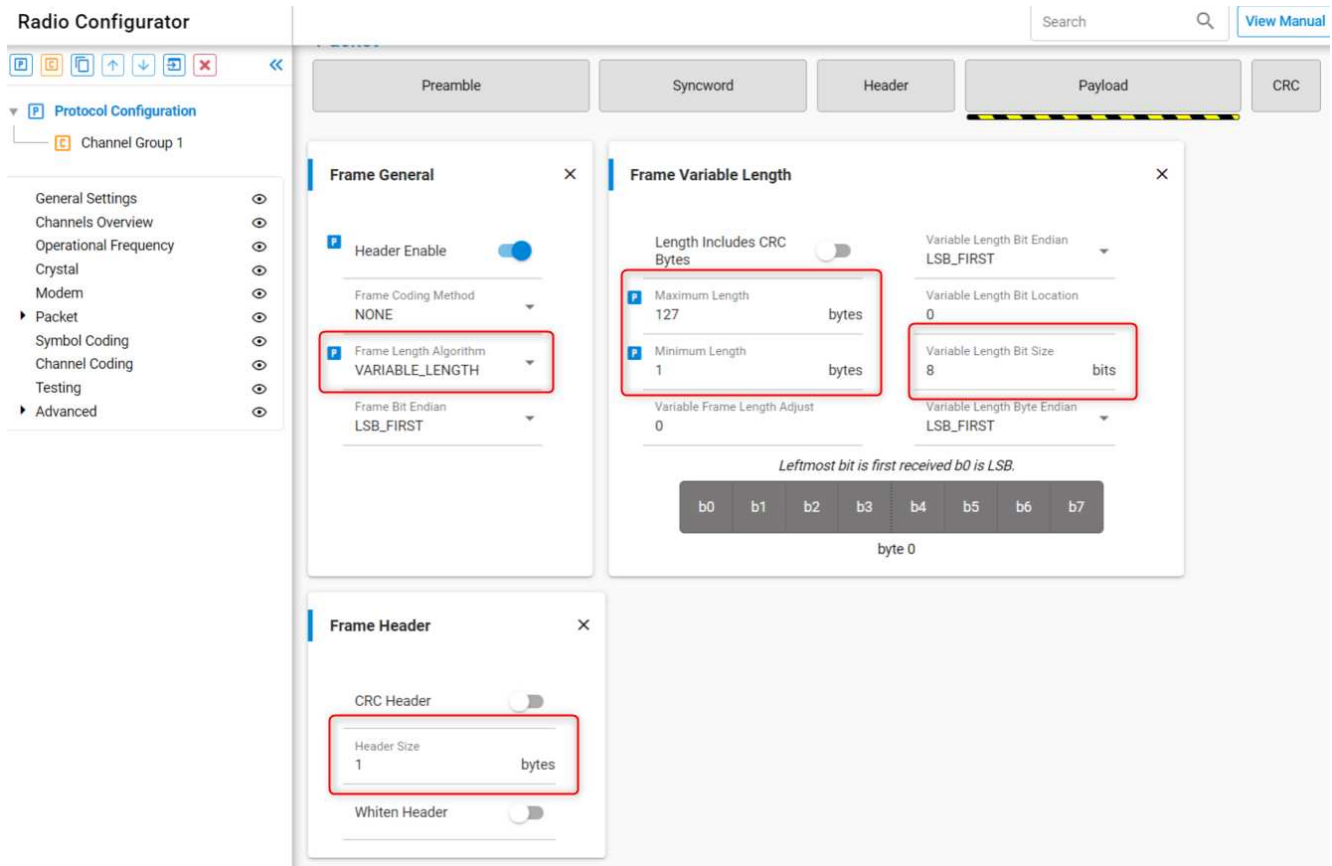
With the API `RAIL_SetFixedLength()`, it's possible to change the pre-configured length (stored in `rail_config.c`) at runtime. Note that this changes the length of the packets on the Rx side as well. If you want to return to the pre-configured value, you can use `RAIL_SetFixedLength(rail_handle, RAIL_SETFIXEDLENGTH_INVALID)`.



Using Variable Length Configurations

A lot of protocols store the length of the packet in the header of the frame. For example, in IEEE 802.15.4, the first byte stores the length (and the maximum is 127). To set this up, use the following settings in the Radio Configurator:

- Set *Frame Length Algorithm* to *VARIABLE_LENGTH*
- This automatically enables the *Header*
- Set the length of the header to *1*
- Set variable length bit size to *8*
- Set the maximum length to *127*



For more information on this setup, and on more advanced variable length configs, see [AN1253](#). Note that the above configuration is not fully IEEE 802.15.4 compatible to make it simpler.

Length decoding works the same way for **both Rx and Tx**. This means that during Tx, we have to make sure that the committed length in the header matches the number of bytes you load into the FIFO. It also means that if we set the length field to more than 127, we will get a transmit error.

This would be a valid 16 bytes frame, both for the above described variable length and 16 bytes fixed length mode, so this is used in the attached sample code:

```
static const uint8_t payload[PAYLOAD_LENGTH] =
{PAYLOAD_LENGTH-1, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
```

We use `PAYLOAD_LENGTH-1` in the length field, since the 1 byte length field itself shouldn't be counted.

The routine `RAIL_SetFixedLength()` is available in variable length mode, and it changes the radio to fixed length operation (again, both for Rx and Tx). Calling `RAIL_SetFixedLength(rail_handle, RAIL_SETFIXEDLENGTH_INVALID)` will restore it to variable length mode.

Setting up the Example

Button Handling

The [linked example](#) sets the tx fifo on init, loads, and sends the packet on any button press. This is implemented using the [Simple Button component](#) on `btn0`, and implementing its callback:

In `app_process.c` (to get access to the button instance, now declared in `autogen/sl_simple_button_instances.h`):

```
#include "sl_simple_button_instances.h"
```

Also add the button callback code (the callback is already declared in `sl_button.h`):

```
void sl_button_on_change(const sl_button_t* handle){  
    if ( sl_button_get_state(handle) == SL_SIMPLE_BUTTON_PRESSED ){  
        send_packet = true;  
    }  
}
```

By default, the Simple Button is configured for a pin which matches the development kit's PB0 button.

Since this callback is in interrupt context, we avoid using RAIL API directly. It is safe in almost all cases, but calling `RAIL_StartTx()` from interrupt context [should be done carefully](#). Instead, we set the volatile `send_packet` variable to true, and call RAIL APIs from `app_process_action()` only:

```
void app_process_action(RAIL_Handle_t rail_handle)  
{  
    if (send_packet) {  
        send_packet = false;  
        // Increment the last byte of the payload  
        payload[SL_TUTORIAL_TRANSMIT_PAYLOAD_LENGTH - 1]++;  
        RAIL_WriteTxFifo(rail_handle,  
            payload,  
            SL_TUTORIAL_TRANSMIT_PAYLOAD_LENGTH,  
            false);  
        RAIL_StartTx(rail_handle,  
            SL_TUTORIAL_TRANSMIT_DEFAULT_CHANNEL,  
            RAIL_TX_OPTIONS_DEFAULT,  
            NULL);  
    }  
}
```

In the above snippet, we're also incrementing the last byte of the payload before every transmit.

Conclusion

We didn't care about possible errors generated by RAIL: we do that [next time](#), which also provides hints for receiving packets.

API Introduced in this Article

Functions

- [RAIL_SetTxFifo\(\)](#)
- [RAIL_WriteTxFifo\(\)](#)
- [RAIL_StartTx\(\)](#)
- [RAIL_SetFixedLength\(\)](#)

Types and enums

- [RAIL_TX_OPTIONS_DEFAULT](#)
- [RAIL_SETFIXEDLENGTH_INVALID](#)

Event Handling

Event Handling and Automatic State Transitions

This tutorial is the continuation of a series started with the article [Introduction](#). The previous chapter was about [transmitting a packet](#).

Please read them first if you haven't.

The completed example for this article is available [on GitHub](#). You can either import this example, as documented in the readme on Github, or follow this article and modify the example from the [previous article](#).

In the previous tutorials, we set up RAIL to do various tasks through API commands. However, RAIL can inform the application on various events almost real time (basically, wiring interrupts through RAIL). So let's improve what we had in the previous tutorial.

The Event Handler Function

You might have already seen this function in `app_process.c` :

```
void sl_rail_app_on_event(RAIL_Handle_t rail_handle, RAIL_Events_t events)
```

This is the event handler function, which is called by the RAIL library, and routed through the **RAIL Utility, Initialization** component.

The `events` variable is a bit field, holding all possible events. Be careful, this bit field does not fit into a 32-bit variable, which is the size of the `int` on EFR32 MCUs.

The event handler should be prepared for multiple events in a single call. There's no "clearFlag" mechanism. Each event will be signaled once, and the application is responsible for handling it.

Keep in mind that almost all events are generated from interrupts, so the event handler is usually running from interrupt context (i.e., interrupts are disabled, unless interrupt priorities are configured). Therefore, the event handler should run fast; if you have to do something that takes more time, set a flag, and do it from the main loop.

You should also be careful with RAIL interrupt safety. Calling state changing APIs (`RAIL_Start*` , `RAIL_Stop*` or `RAIL_Idle`) from the interrupt handler is allowed, but we recommend to read [the interrupt safety article](#) before doing so.

The events enabled by default are configured on the **RAIL Utility, Initialization** component (under **Platform/Radio**, press **Configure**):

RAIL Utility, Initialization (inst0)

[View Source](#)

Radio Event Configuration

RX Radio Events

RX Timing Detect



RX Timing Lost



RX Preamble Detect



RX Preamble Lost



RX Sync1 Detect



RX Sync2 Detect



RX Filter Passed



RX Address Filtered



RX Packet Received



RX Frame Error



RX Packet Aborted



RX Timeout



RX Channel Hopping Complete



RX Duty Cycle RX End



TX Radio Events

TX Packet Sent



TX Aborted



TX Blocked



TX Started



RSSI Radio Events

RSSI Average Done



FIFO Radio Events

RX FIFO, Almost Full



RX FIFO, Full



RX FIFO, Overflow



TX FIFO, Almost Empty



TX FIFO, Underflow



However, using the RAIL API directly is actually simpler than using the event configuration capabilities of this component. The API for this is `RAIL_ConfigEvents()`. You can keep the event setup enabled; it doesn't really matter.

`RAIL_ConfigEvents()` has a `mask` and an `events` parameter, making it simple to configure multiple events at the same time; e.g., to enable or disable a single event, leaving the others untouched:

```
//disable RAIL_EVENT_TX_PACKET_SENT
RAIL_ConfigEvents(railHandle, RAIL_EVENT_TX_PACKET_SENT, 0);

//enable RAIL_EVENT_TX_PACKET_SENT
RAIL_ConfigEvents(railHandle, RAIL_EVENT_TX_PACKET_SENT, RAIL_EVENT_TX_PACKET_SENT);
```

RAIL also defines a few event groups (defined in `railtypes.h`, the most useful are:

- `RAIL_EVENTS_ALL` includes all the events
- `RAIL_EVENTS_NONE` includes no events (equivalent to 0)
- `RAIL_EVENTS_TX_COMPLETION` includes all possible events after a successful `RAIL_StartTx`
- `RAIL_EVENTS_RX_COMPLETION` includes all possible events that can result in state transition from Rx mode

This can be used, for example, to disable or enable group of events:

```
//disable all events
RAIL_ConfigEvents(railHandle, RAIL_EVENTS_ALL, 0);

//enable all events
RAIL_ConfigEvents(railHandle, RAIL_EVENTS_TX_COMPLETION, RAIL_EVENTS_TX_COMPLETION);
```

If you enable/disable events in group, it's a good practice to create similar macro group of events in your application.

Note that enabling all events is not recommended. Some events - depending on the radio config - might happen very often, and even if you don't do anything on that event in the handler, just the execution of the event handler takes some CPU time.

RAIL has no API to flush pending events, but disabling and enabling them will have that result:

```
//flush RAIL_EVENT_TX_PACKET_SENT
RAIL_ConfigEvents(railHandle, RAIL_EVENT_TX_PACKET_SENT, 0);
RAIL_ConfigEvents(railHandle, RAIL_EVENT_TX_PACKET_SENT, RAIL_EVENT_TX_PACKET_SENT);
```

However, you only need to do this in rare cases. Enabling events is not like unmasking interrupts. Enabling them will not trigger the event handler on past events.

Tx Error Handling

In the last tutorial, we sent out a frame, but we didn't check the results because it would have needed the event handler. Let's do the following:

- Toggle led0 on success (Tx or Rx)
- Toggle led1 on error (StartTX, Rx or Tx error)

Nb: at startup, all leds are set off by the stack.

First, for the LEDs, enable the [Simple LED](#) component with *led0* and *led1* instances.

In `app_process.c` (to get access to the led instances, now declared in `autogen/sl_simple_led_instances.h`):

```
#include "sl_simple_led_instances.h"
```

For the actual radio related changes, we'll need to enable the success and error events. We can do that during init since we don't really need to disable anything at runtime for such simple applications:

```
RAIL_Handle_t app_init(void)
{
    //...
    RAIL_ConfigEvents(rail_handle, RAIL_EVENTS_ALL, RAIL_EVENTS_TX_COMPLETION);
}
```

Next, let's check for errors returned by the StartTx call:

```
RAIL_Status_t status = RAIL_StartTx(rail_handle, 0, RAIL_TX_OPTIONS_DEFAULT, NULL);
if ( status != RAIL_STATUS_NO_ERROR ) {
    sl_led_toggle(&sl_led_led1);
}
```

Finally, let's check the events for success and errors:

```
void sl_rail_util_on_event(RAIL_Handle_t rail_handle, RAIL_Events_t events)
{
    (void) rail_handle;
    if ( events & RAIL_EVENTS_TX_COMPLETION ) {
        if ( events & RAIL_EVENT_TX_PACKET_SENT ) {
            sl_led_toggle(&sl_led_led0);
        } else {
            sl_led_toggle(&sl_led_led1); //all other events in RAIL_EVENTS_TX_COMPLETION are errors
        }
    }
}
```

With this, you should see led0 toggling on every transmitted packet.

Note that the example on Github has more error handling added to it, e.g., the return value of `RAIL_SetTxFifo()` is also checked.

Receiving Packets

Let's add receive capabilities to this application! Again, use led0 to report success and led1 for errors.

First, we need to modify the init code.

In `app_process.c` (to get access to the led instances):

```
#include "sl_simple_led_instances.h"
```

We need Rx success/error events enabled, and we need to start the radio in Rx mode:

```
RAIL_Handle_t app_init(void)
{
    //...

    RAIL_Status_t status = RAIL_ConfigEvents(rail_handle,
                                             RAIL_EVENTS_ALL,
                                             RAIL_EVENTS_TX_COMPLETION
                                             | RAIL_EVENTS_RX_COMPLETION);
    if (status != RAIL_STATUS_NO_ERROR) {
        // Turn led1 ON on ConfigEvents Error
        sl_led_toggle(&sl_led_led1);
    }
    status = RAIL_StartRx(rail_handle,
                          0,
                          NULL);
    if (status != RAIL_STATUS_NO_ERROR) {
        // Turn led1 ON on StartRx Error
        sl_led_toggle(&sl_led_led1);
    }
}
```

The API `RAIL_StartRx()` is similar to Tx. The second argument of the function indicates which channel is going to be used.

Finally, let's check the events for success and errors (in `app_process.c`):

```
void sl_rail_util_on_event(RAIL_Handle_t rail_handle, RAIL_Events_t events)
{
    //...
    if (events & RAIL_EVENTS_RX_COMPLETION) {
        if (events & RAIL_EVENT_RX_PACKET_RECEIVED) {
            sl_led_toggle(&sl_led_led0);
        } else {
            sl_led_toggle(&sl_led_led1); //all other events in RAIL_EVENTS_RX_COMPLETION are errors
        }
    }
}
```

Automatic State Transitions

By default, after receiving or transmitting a packet, RAIL will switch to idle mode (i.e., turn off the radio). Since we want it to keep receiving future packets, we can use RAIL's auto state transition feature. This can be configured on the *RAIL Utility*, *Initialization* component, let's set all dropdowns to *RX*:

Auto Transition Configuration

Enable Automatic Radio State Transitions



TX Transitions

Transition on TX Success

RX

Transition on TX Error

RX

RX Transitions

Transition on RX Success

RX

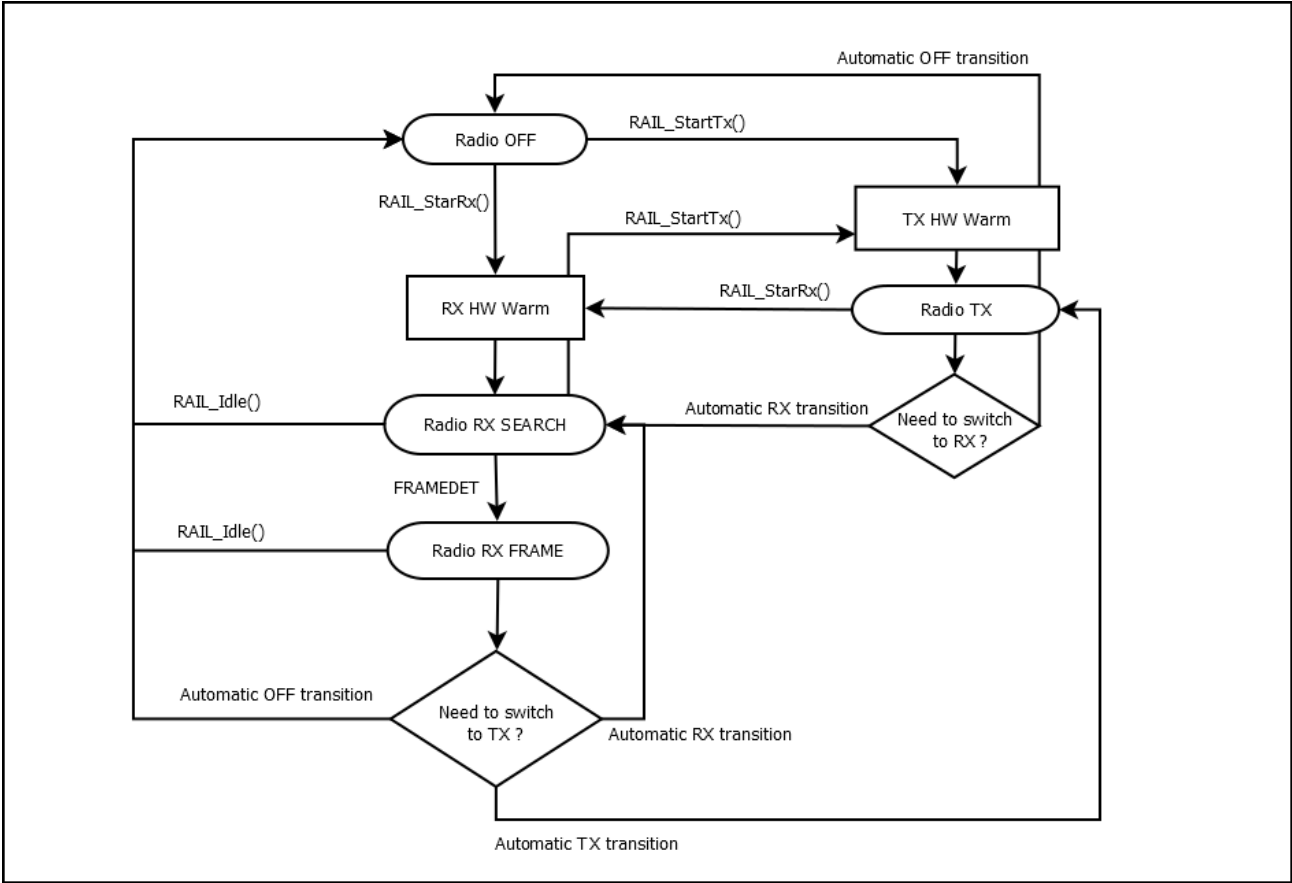
Transition on RX Error

RX

We're basically saying that after the packets are received/transmitted, we want to return to Rx. These configurations can be also changed in run-time, using the `RAIL_SetRxTransitions()` and `RAIL_SetTxTransitions()` APIs.

If you compile and flash the program after this modification, you should see that now both WSTK toggles led0 on every button press.

The following illustrates a high level flowchart of the radio controller:



Note that in the Tx state, calling `RAIL_Idle()` will operate a transition to the radio OFF state.

Testing and Conclusion

If you install this example to two boards (make sure you attach the antennas if you use sub-GHz kits on sub-GHz config), you will see that on the press of btn0 on either device, led0 on both devices toggle. However, we can't see what we're receiving, since we never actually download the messages.

We're going to fix this [next time](#).

API Introduced in this Article

Functions

- [RAIL_ConfigEvents\(\)](#)
- [RAIL_StartRx\(\)](#)
- [RAIL_SetRxTransitions\(\)](#)
- [RAIL_SetTxTransitions\(\)](#)

Types and Enums

- [RAIL_Events_t](#)
- [RAIL_EVENTS_ALL](#)
- [RAIL_EVENTS_TX_COMPLETION](#)
- [RAIL_EVENTS_RX_COMPLETION](#)
- [RAIL_EVENTS_NONE](#)
- [RAIL_EVENT_TX_PACKET_SENT](#)
- [RAIL_EVENT_RX_PACKET_RECEIVED](#)

Downloading Messages

Downloading Received Messages

This tutorial is the continuation of a series started with the article [Introduction](#). The previous chapter was about [Event Handling and Automatic State Transitions](#).

Please read them first if you haven't.

The completed example for this article is available on [GitHub](#). You can either import this example, as documented in the readme on Github, or follow this article and modify the example from the [previous article](#).

In the previous tutorial, we had both transmit and receive working, but we couldn't see what we received. This tutorial will solve that problem.

We're also adding calibrations, which is recommended for all projects.

You can find a SimpleTRX example in Simplicity Studio, which has a similar purpose. However, the result of this tutorial will be somewhat different. Mainly, this tutorial is not very pedantic on checking errors returned by all functions, to make it easier to understand. However, we highly recommend checking all possible errors, therefore using the SimpleTRX example in Studio as a reference.

Enabling Calibrations

All calibrations are enabled by default in **RAIL Utility, Initialization** as depicted below, but require some application code to perform them.

RAIL Utility, Initialization (inst0)

[View Source](#)

RAIL Init Configuration

Initialize RAIL

Project Configuration

Enable Radio Configurator Support (include rail_config.c/h generated by Radio Config)



Enable Bluetooth Support



Enable Scheduler Support (for DMP)



Enable RAIL Init Complete Callback



Protocol Configuration

Radio Configuration

Default Radio Configuration

Proprietary (use rail_config.c/h)

Proprietary Radio Config Index (if Proprietary radio config selected, use index X of channelConfigs[X])



0



Calibration Configuration

Calibration Notifications

Enable Temperature Calibration Notifications (RAIL_EVENT_CAL_NEEDED radio event issued when temperature calibrations needed, for example VCO calibration)



Enable One-time Calibration Notifications (RAIL_EVENT_CAL_NEEDED radio event issued when one-time calibrations needed, for example IR calibration)



First, we need to add the related event during initialization to enable it (in `app_init.c`):

```
RAIL_Handle_t app_init(void)
{
    //...
    RAIL_Status_t status = RAIL_ConfigEvents(rail_handle, RAIL_EVENTS_ALL,
        RAIL_EVENTS_TX_COMPLETION
        | RAIL_EVENTS_RX_COMPLETION
        | RAIL_EVENT_CAL_NEEDED
    );
    //...
}
```

Next, we need to perform calibrations when RAIL requests them (in `app_process.c`):

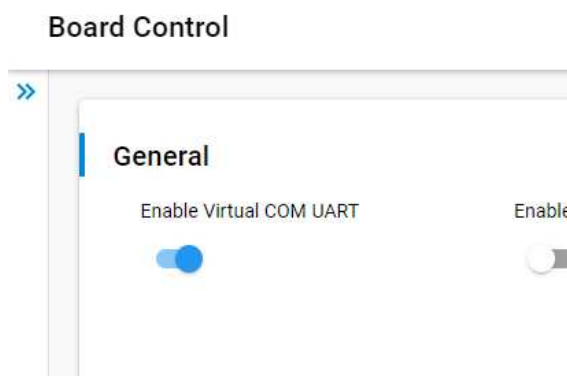
```
void sl_rail_util_on_event(RAIL_Handle_t rail_handle, RAIL_Events_t events)
{
    //...
    if (events & RAIL_EVENT_CAL_NEEDED) {
        RAIL_Status_t status = RAIL_Calibrate(rail_handle,
            NULL,
            RAIL_CAL_ALL_PENDING);
        if (status != RAIL_STATUS_NO_ERROR) {
            slLed_toggle(&slLed_led1);
        }
    }
}
```

This is all that's needed from an application to perform all calibrations available on EFR32, to have the maximum available performance. Calibrations are detailed further in [another tutorial](#).

Preparing for Printing

In the previous tutorials, we already enabled LEDs and buttons, but we'll need a more user friendly output: We need a serial console. To enable that, install the **Log** component under **Application/Utility**. This should pull up its dependencies, including an **IO Stream: EUSART** or **UART** component, which is the serial driver itself.

Finally, we need to enable **Virtual COM UART** on the **Board Control** component:



This pulls up a GPIO, which will enable the USB-UART bridge on the development kit (on the motherboard, if you use a dual board development kit).

Receive FIFO

We already configured Tx FIFO in multiple ways. RAIL also implements an Rx FIFO. By default, it's 512 Bytes, allocated by the RAIL library. We will use that, but it is configurable. See `RAILCb_SetupRxFifo()` and `RAIL_SetRxFifo()` in the [API documentation](#) for details.

By default, you cannot use the whole 512 Bytes, as RAIL adds a few bytes to each packet to store timestamp, RSSI, and similar information in the FIFO (sometimes we call this "appended info").

We're working on what we call packet mode. It's the simpler, and recommended way, but it's not possible to handle packets bigger than the receive FIFO with this method, in which case FIFO mode should be used. We return to that topic in a [later tutorial](#).

The Packet Handle

The FIFO is accessible with `RAIL_RxPacketHandle_t` variables or packet handles. Obviously, we don't have the handle of the packet we just received, but we have "sentinel" handles:

- `RAIL_RX_PACKET_HANDLE_NEWEST`
- `RAIL_RX_PACKET_HANDLE_OLDEST`
- `RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE`
- `RAIL_RX_PACKET_HANDLE_INVALID`

Note that `RAIL_RX_PACKET_HANDLE_OLDEST` and `RAIL_RX_PACKET_HANDLE_NEWEST` will always return with packet information. For example, OLDEST will return with one of the following:

1. the oldest packet received, if available
2. the packet the radio is receiving at the moment, if available and if no fully received packet is available
3. the placeholder which will be used for the next packet the radio will receive, if no fully or partially received packet is available

If you don't care about upcoming or ongoing packets, you can use `RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE`. It will only return with fully received packets.

When to Read the FIFO

Let's say we want to download the packet when it's fully received (although it's possible to access it during reception). We **must** let RAIL know this in the event `RAIL_EVENT_RX_PACKET_RECEIVED`. If we don't, **RAIL automatically frees the memory allocated to the packet we just received.**

We have two options:

- Download the packet from the event handler
- Hold the packet in the FIFO and download later

While the first one is simpler, it is desirable in most cases to keep large memory copy operations outside of interrupt handlers, therefore the attached example demonstrates the second method.

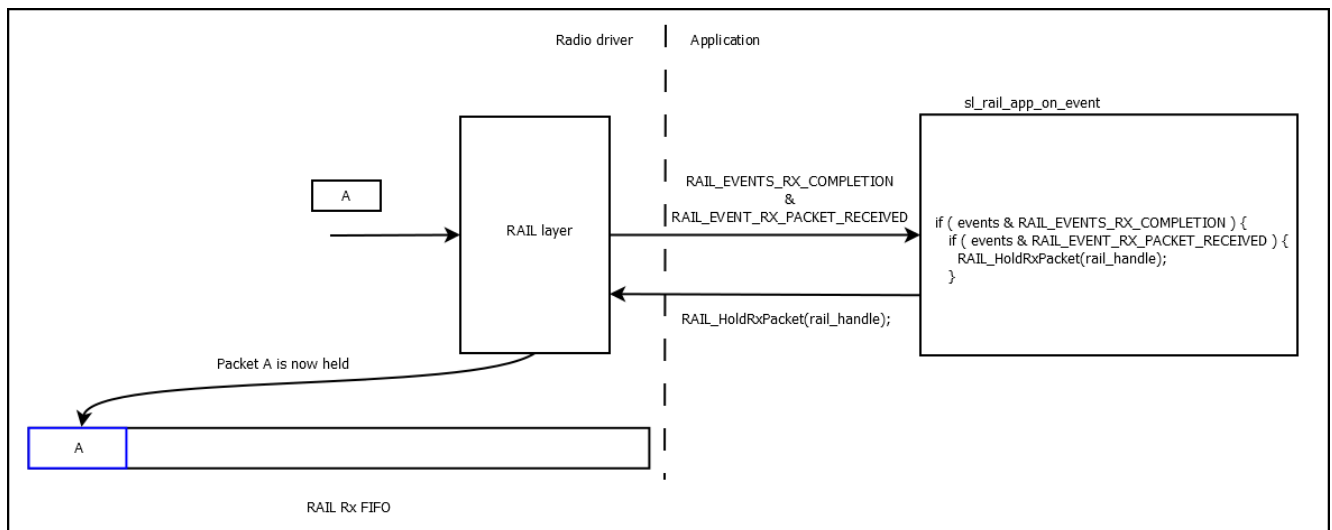
Note however, that if you plan to use [dynamic multiprotocol, DMP](#), you must read the FIFO before yielding the radio. On every **DMP** protocol change, RAIL resets the receive FIFO. In this case, it's usually better to download the packet in the event handler.

The Event Handler

In the event handler, we want to instruct RAIL to hold all successfully received frames:


```
void sl_rail_util_on_event(RAIL_Handle_t rail_handle, RAIL_Events_t events)
{
    //...
    if (events & RAIL_EVENTS_RX_COMPLETION) {
        if (events & RAIL_EVENT_RX_PACKET_RECEIVED) {
            if (RAIL_HoldRxPacket(rail_handle) == RAIL_RX_PACKET_HANDLE_INVALID) {
                sl_led_toggle(&sl_led_led1);
            }
            sl_led_toggle(&sl_led_led0);
        } else {
            sl_led_toggle(&sl_led_led1); // all other events in
            // RAIL_EVENTS_RX_COMPLETION are errors
        }
    }
}
```

The API `RAIL_HoldRxPacket` will also return a packet handler, but we only report an error if it did not. We don't need the handle itself, since we're going to use sentinel handles. **Keep in mind that you *must* release all packets that were held; otherwise, you will lose part of your Rx fifo, essentially leaking memory.** Note also that this API shouldn't be used when developing for dynamic multiprotocol (DMP).



Download and Print

First, let's create a buffer we can use to download the packet:

```
static uint8_t rx_buffer[256];
```

In `app_process_action()`, if there's something in the packet handle, let's download and print it:

```

void app_process_action(RAIL_Handle_t rail_handle)
{
    static RAIL_RxPacketHandle_t packet_handle;
    static RAIL_RxPacketInfo_t packet_info;
    static RAIL_RxPacketDetails_t packet_details;
    //...
    packet_handle = RAIL_GetRxPacketInfo(rail_handle,
                                         RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE,
                                         &packet_info);
    if (packet_handle != RAIL_RX_PACKET_HANDLE_INVALID) {
        RAIL_CopyRxPacket(rx_buffer, &packet_info);
        RAIL_Status_t status =
            RAIL_GetRxPacketDetails(rail_handle, packet_handle, &packet_details);
        if (status != RAIL_STATUS_NO_ERROR) {
            sl_led_toggle(&sl_led_led1);
        }
        status = RAIL_ReleaseRxPacket(rail_handle, packet_handle);
        if (status != RAIL_STATUS_NO_ERROR) {
            sl_led_toggle(&sl_led_led1);
        }

        app_log("RX");
        for (int i = 0; i < packet_info.packetBytes; i++) {
            app_log(" 0x%02X", rx_buffer[i]);
        }
        app_log(" RSSI=%d dBm\n", packet_details.rssi);
    }
}

```

Let's go through this, line by line:

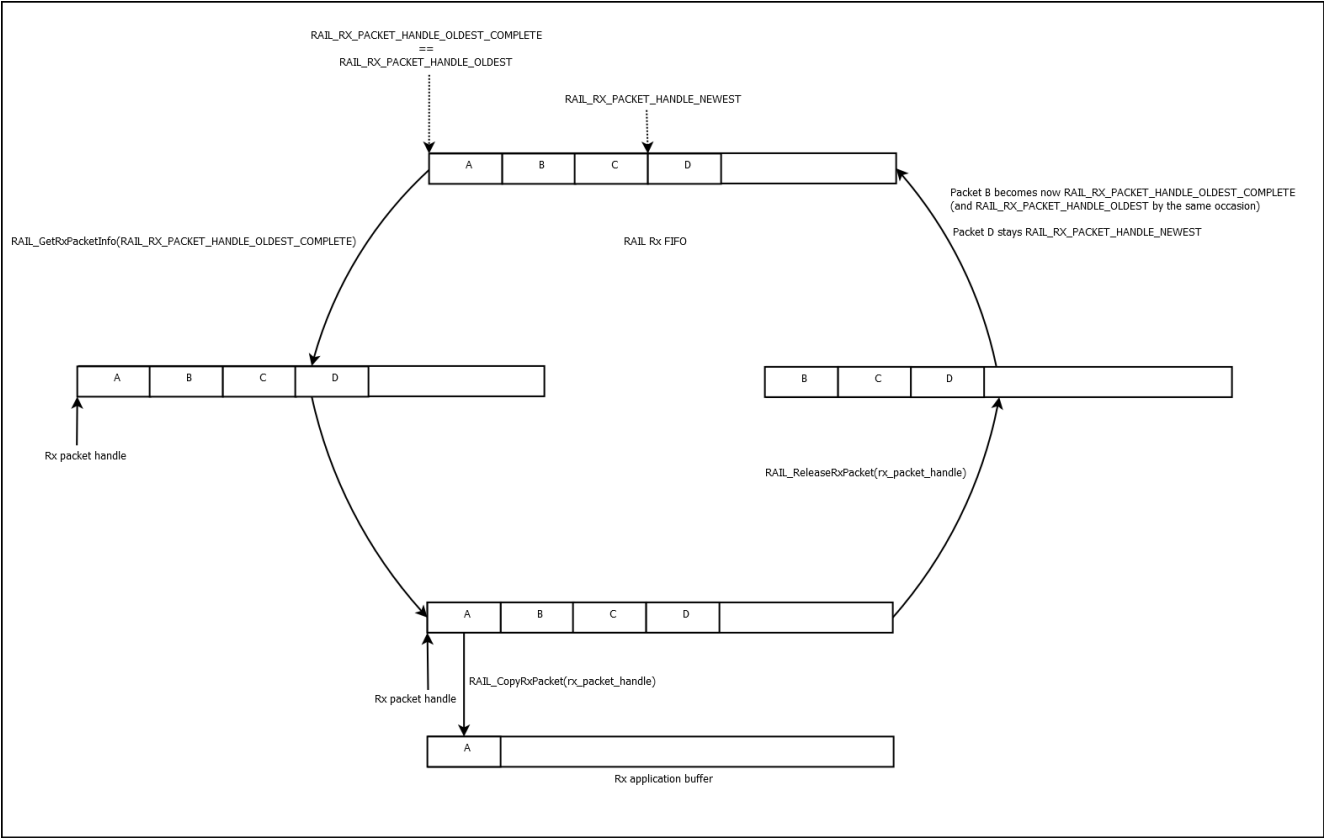
First, `RAIL_GetRxPacketInfo()` will return with a valid `packet_handle` if we have a completely received frame. In that case, it also returns a usable `packet_info` which has the length of the received packet, and pointers to download it. Note that the length will include the full frame, and is always valid, whichever length configuration is used (fixed or variable).

Since the receive buffer is a ring buffer, the packet in the buffer might not be in a single sequential buffer, copying from that using `memcpy()` is a bit complicated: `RAIL_CopyRxPacket()` macro implements that.

The API `RAIL_GetRxPacketDetails()` will return some useful information of the packet, such as RSSI or timestamps.

Finally, we release the packet, using `RAIL_ReleaseRxPacket()`.

Note that this method is safe, even if we receive a new packet while downloading one. That packet will be downloaded and printed when `app_process_action()` is called the next time.



It is important to note here that there is a subtle difference between `RAIL_PACKET_HANDLE_OLDEST` and `RAIL_PACKET_HANDLE_OLDEST_COMPLETE`.

`RAIL_PACKET_HANDLE_OLDEST` as well as `RAIL_PACKET_HANDLE_NEWEST` become valid at beginning of frame reception. As a result, it cannot be used for packets reception easily in an active loop.

The following tab clarifies the differences between the handles:

	NEWEST	OLDEST	OLDEST_COMPLETE
FIFO Empty	INVALID	INVALID	INVALID
FIFO receiving packet A	Points to packet A	Points to packet A	INVALID
FIFO has received packet A	Points to packet A	Points to packet A	Points to packet A
FIFO receiving packet B	Points to packet B	Points to packet A	Points to packet A
FIFO receiving packet B and A is copied/released	Points to packet B	Points to packet B	INVALID

While this isn't exhaustive, it helps to clarify the ambiguity between all three handles.

Download in the Event Handler

Downloading the packet in the event handler is essentially the same as what we did in the main loop above, except we don't need to worry about releasing:

```
if (events & RAIL_EVENT_RX_PACKET_RECEIVED) {
    RAIL_RxPacketInfo_t packet_info;
    RAIL_RxPacketHandle_t packet_handle = RAIL_GetRxPacketInfo(rail_handle, RAIL_RX_PACKET_HANDLE_NEWEST, &packet_info);
    RAIL_CopyRxPacket(rx_buffer, &packet_info);
    RAIL_Status_t status = RAIL_GetRxPacketDetails(rail_handle, packet_handle, &packet_details);
}
```

Note that `packet_details` should be a global variable (i.e. declared outside of the functions to be available to both functions) in this case.

Conclusion

With this, you can use RAIL for basic transmit and receive, which concludes the getting started series. However, RAIL can do much more. You can continue with the basic tutorials.

API Introduced in this Article

Functions

- [RAIL_Calibrate\(\)](#)
- [RAILCb_SetupRxFifo\(\)](#)
- [RAIL_SetRxFifo\(\)](#)
- [RAIL_HoldRxPacket\(\)](#)
- [RAIL_ReleaseRxPacket\(\)](#)
- [RAIL_GetRxPacketInfo\(\)](#)
- [RAIL_GetRxPacketDetails\(\)](#)
- [RAIL_CopyRxPacket\(\)](#)

Types and Enums

- [RAIL_RxPacketHandle_t](#)
- [RAIL_RxPacketInfo_t](#)
- [RAIL_RxPacketDetails_t](#)
- [RAIL_EVENT_CAL_NEEDED](#)
- [RAIL_RX_PACKET_HANDLE_NEWEST](#)
- [RAIL_RX_PACKET_HANDLE_OLDEST](#)
- [RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE](#)
- [RAIL_RX_PACKET_HANDLE_INVALID](#)

Scheduling and Timestamping

Time, Timestamping, and Scheduling

Timing is an important aspect of most wireless protocols. While it's possible to use a generic timer for most of the required tasks, Silicon Labs' Wireless MCUs provide a protocol timer, which works together with the radio state machine, providing many advantages. This protocol timer is the base of all RAIL timing features.

Using the protocol timer, you can completely eliminate interrupt latencies, and you can get more accurate and more repeatable timing, even if you update the SDK version. Using the protocol timer in dynamic multiprotocol (DMP) is highly recommended as it tells the radio scheduler what you are planning to use the radio for.

RAIL Time

RAIL timebase (or the resolution of RAIL time) is 1 μ s regardless of the oscillator's frequency, counted from boot. The same timebase is used for everything (timestamping and absolute timers). It's stored in `RAIL_Time_t`, currently implemented as `uint32_t`, what means that it wraps every 1.19 hours.

Note that different hardware might run at different speeds, which is converted to this universal 1 μ s timebase. For this reason, you should never compare RAIL times with equality operators (`==` and `!=`) as it might skip values.

RAIL provides the usual getter/setter for this:

```
RAIL_Time_t RAIL_GetTime();
RAIL_Status_t RAIL_SetTime(RAIL_Time_t time);
```

Note that it's generally a bad practice to use `RAIL_SetTime()`: It can cause serious issues when timers are running. Also, RAIL time is one of the few shared resources between protocols in dynamic multiprotocol mode.

Timestamps

By default, each Tx and Rx packet is timestamped with the RAIL time on a fixed position of the packet.

While it would be possible to save the value of a timer at a specific RAIL event (like `RAIL_EVENT_TX_PACKET_SENT`, `RAIL_EVENT_RX_SYNC1_DETECT`, `RAIL_EVENT_RX_PACKET_RECEIVED`, etc.) this isn't recommended.

The reason is that at the RAIL API level, there's no way to guarantee time accuracy between the transmission/reception routine is called and the event is raised. Moreover, this would not be possible if radio interrupts are disabled.

Acquire Tx Timestamp

To get the timestamp of transmitted packets, use `RAIL_GetTxPacketDetails()` or - for smaller code size - `RAIL_GetTxPacketDetailsAlt2()` in the context of the `RAIL_EVENT_TX_PACKET_SENT` event:

```
if ( events & RAIL_EVENT_TX_PACKET_SENT ){
    RAIL_TxPacketDetails_t packetDetails;
    packetDetails.isAck = false;
    packetDetails.timeSent.timePosition = RAIL_PACKET_TIME_AT_PACKET_END;
    packetDetails.timeSent.totalPacketBytes = 16;
    RAIL_GetTxPacketDetails(railHandle, &packetDetails);
}
```

When using this API take care of the value passed by the `packetDetails.timeSent.totalPacketBytes`. It must account for all bytes sent over the air after the Preamble and Sync word(s) including payload CRC bytes and a potential header (length). If

the packet is coded, e.g., with Manchester or FEC, it could also mean that the packet over the air is longer than the packet length reported by RAIL. In this case, it is recommended to validate the timestamp accuracy.

The same timestamp can be acquired by using the alternative API:

```
if ( events & RAIL_EVENT_TX_PACKET_SENT ){
    RAIL_TxPacketDetails_t packetDetails;
    packetDetails.isAck = false;
    RAIL_GetTxPacketDetailsAlt2(railHandle, &packetDetails);
    // ...
    // Timestamp position discussed later
}
```

Though, there is a [RAIL_GetTxPacketDetailsAlt\(\)](#) API, we don't recommend to use it because it has a more complicated argument list.

Note that in case of using [RAIL_GetTxPacketDetails\(\)](#) the fields initialized in `packetDetails` before passing it to the function. The first two (`isAck` and `timeSent.timePosition`) **must always be initialized** to get proper timestamp; `timeSent.totalPacketBytes` is not mandatory, but not all time positions work without it.

For the alternate API, you only have to know if the packet is ACK or not, but you have to convert the timestamp later.

We return to the time positions soon, but first, let's see how to get the timestamp of a received packet.

Acquire Rx Timestamp

Similarly to Tx, RAIL defines [RAIL_GetRxPacketDetails\(\)](#) and [RAIL_GetRxPacketDetailsAlt\(\)](#) APIs for Rx, and you should use them quite similarly:

```
RAIL_RxPacketInfo_t packetInfo;
RAIL_RxPacketDetails_t packetDetails;
RAIL_RxPacketHandle_t packetHandle = RAIL_GetRxPacketInfo(railHandle, RAIL_RX_PACKET_HANDLE_NEWEST, &packetInfo);
packetDetails.timeReceived.timePosition = RAIL_PACKET_TIME_AT_SYNC_END;
packetDetails.timeReceived.totalPacketBytes = packetInfo.packetBytes + 2; /* 2 Byte CRC is used on this example setup, what we need to
calculate in here. */
RAIL_GetRxPacketDetails(railHandle, packetHandle, &packetDetails);
```

Or using the alternative API:

```
RAIL_RxPacketInfo_t packetInfo;
RAIL_RxPacketDetails_t packetDetails;
RAIL_RxPacketHandle_t packetHandle = RAIL_GetRxPacketInfo(railHandle, RAIL_RX_PACKET_HANDLE_NEWEST, &packetInfo);
RAIL_GetRxPacketDetailsAlt(railHandle, packetHandle, &packetDetails);
```

Just like for Tx, `timeReceived.timePosition` **must always be initialized** to get a proper timestamp, while `timeReceived.totalPacketBytes` is not mandatory, but not all time positions work without it. However, it's relatively simple to get the length of a received packet, so there's not much gain avoiding it.

The alternative API doesn't need any initialization, but requires conversion later. Note that a `packetHandle` was used to identify the packet we need to be timestamped. This means that the timestamp of the received packet is stored as long as the packet is not released either automatically or by calling [RAIL_ReleaseRxPacket\(\)](#) after [RAIL_HoldRxPacket\(\)](#). As a result, these routines do not need to be called in the RAIL event handler (interrupt context).

This means that the Tx timestamps can only be obtained in the RAIL event handler (interrupt context), as opposed to the RX timestamps, which persist until the received packets are released.

Timestamp Positions

Position Conversions with Time Positions

The following `RAIL_PacketTimePosition_t` time positions are available for timestamps:

- `RAIL_PACKET_TIME_AT_PREAMBLE_START`
- `RAIL_PACKET_TIME_AT_SYNC_END`
- `RAIL_PACKET_TIME_AT_PACKET_END`
- `RAIL_PACKET_TIME_DEFAULT`

The first three are self-explanatory. `RAIL_PACKET_TIME_DEFAULT` is the hardware implementation, which never needs the `totalPacketBytes`. If default was passed, the function will update the `timePosition` to one of the first three, which is the default on the hardware.

On EFR32 Series 1, this will be `RAIL_PACKET_TIME_AT_SYNC_END` for Rx and `RAIL_PACKET_TIME_AT_PACKET_END` for Tx packet.

On EFR32 Series 2, `RAIL_PACKET_TIME_AT_PACKET_END` is used for both Tx and Rx.

In some cases, `totalPacketBytes` is required to convert the default timestamp position to the requested one. The function will update `timePosition`, if the `totalPacketBytes` was used on the following:

- `RAIL_PACKET_TIME_AT_PREAMBLE_START_USED_TOTAL`
- `RAIL_PACKET_TIME_AT_SYNC_END_USED_TOTAL`
- `RAIL_PACKET_TIME_AT_PACKET_END_USED_TOTAL`

This can be used to recognize an inaccurate timestamp, if `totalPacketBytes` is unknown.

Position Conversions with the Alternative APIs

The alternative APIs store the timestamp for a non-defined position. To convert it to timestamp in a given position, one of the following APIs must be called:

```
//Rx APIs:
RAIL_Status_t RAIL_GetRxTimeFrameEndAlt(RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails);
RAIL_Status_t RAIL_GetRxTimePreambleStartAlt(RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails);
RAIL_Status_t RAIL_GetRxTimeSyncWordEndAlt(RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails);
//Tx APIs:
RAIL_Status_t RAIL_GetTxTimeFrameEndAlt(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails);
RAIL_Status_t RAIL_GetTxTimePreambleStartAlt(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails);
RAIL_Status_t RAIL_GetTxTimeSyncWordEndAlt(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails);
```

These alternative conversion APIs all need the `pPacketDetails` previously used by the `RAIL_GetTxPacketDetailsAlt2()` or `RAIL_GetRxPacketDetailsAlt()` APIs as an input parameter.

Advantages of Using the Alternative APIs

The alternative timestamping APIs may have some benefit over the regular `RAIL_GetTxPacketDetails()` and `RAIL_GetRxPacketDetails()` APIs:

- If you don't want to convert the timestamp in interrupt context (i.e., the event handler) in case of Rx, or you don't want to access some variables, like the packet length from interrupt context, you can choose the alternative functions.
- You might only want timestamps on specific packets, depending on the payload. Using the alternate API can save CPU time and RAM as you only do the complex timestamp conversion on those specific packets.
- You can obtain a packet timestamp on an ongoing transmitted packet utilizing `RAIL_GetTxTimePreambleStartAlt()` handling an `RAIL_EVENT_TX_STARTED` event. Note that this event does not apply to ACK transmits (currently).
- Also, these are useful, if you want to reduce code size.

General Purpose Timer

By default, you can set up a single general purpose timer on the RAIL timebase. The following APIs are usable for this timer (for the detailed arguments, please check the API documentation):

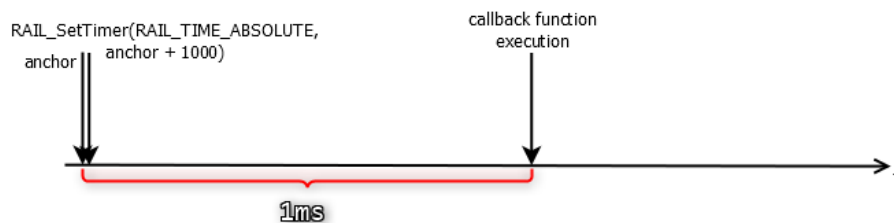
- [RAIL_SetTimer](#)(RAIL_Handle_t, RAIL_Time_t, RAIL_TimeMode_t, RAIL_TimerCallback_t)
- [RAIL_GetTimer](#)(RAIL_Handle_t)
- [RAIL_CancelTimer](#)(RAIL_Handle_t)
- [RAIL_IsTimerExpired](#)(RAIL_Handle_t)
- [RAIL_IsTimerRunning](#)(RAIL_Handle_t)

There are a few parameters that require explanation. [RAIL_TimerCallback_t](#) is a function pointer type to the function which will be called when the timer fires. [RAIL_TimeMode_t](#) can take two values in this case:

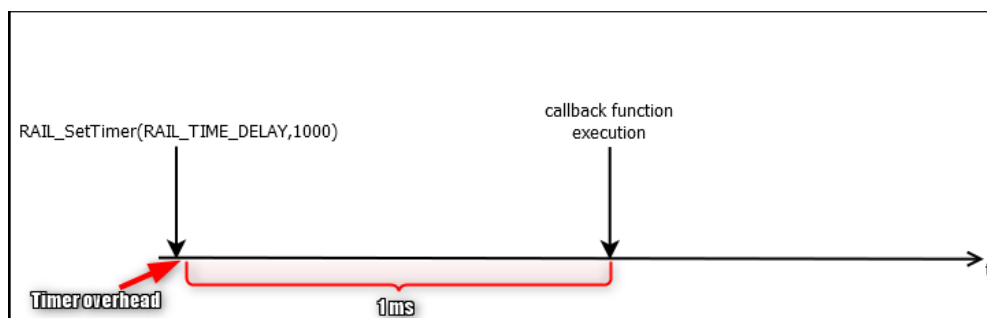
- [RAIL_TIME_ABSOLUTE](#)
- [RAIL_TIME_DELAY](#)

If [RAIL_TIME_ABSOLUTE](#) is used, RAIL interprets the timing in radio time base absolute time. On the other hand, if [RAIL_TIME_DELAY](#) is used, the time specified is a delay starting from the the function call in the RAIL timebase. However, [RAIL_TIME_DELAY](#) only guarantees that *at least* the specified delay is elapsed; it does not compensate the API processing time. If you need accurate timing, you should use [RAIL_TIME_ABSOLUTE](#) :

```
RAIL_Time_t anchor = RAIL_GetTime();
RAIL_SetTimer(railHandle, anchor + 1000, RAIL_TIME_ABSOLUTE, timerCb);
```



You can notice that the timer starts from the anchor and not from the [RAIL_SetTimer\(\)](#) function call. If [RAIL_TIME_DELAY](#) were used, this is how it would look like, with the small processing time added to the delay:



Note though, that this processing time is consistent in the same environment. So if for some reason, you cannot get an anchor point, you can use a [RAIL_TIME_DELAY](#), measure it, and compensate from the application. However, if you update RAIL, or change the part the code is running on, the delay might change.

Restarting timers (i.e., to change the timeout) is not allowed. If the timer might be running when you set it, you should call [RAIL_CancelTimer\(\)](#) first.

Multi-timer

RAIL provides timer virtualization over RAIL timer. In single protocol mode, it's not enabled to save code space. To enable it, call [RAIL_ConfigMultiTimer\(\)](#) with `true` argument.

Multi-timer is always enabled in dynamic multiprotocol (DMP) mode.

It provides basically the same APIs and the same precision as the regular RAIL timer:

- [RAIL_SetMultiTimer](#)(RAIL_MultiTimer_t*, RAIL_Time_t, RAIL_TimeMode_t, RAIL_MultiTimerCallback_t, void*)
- [RAIL_GetMultiTimer](#)(RAIL_MultiTimer_t*)
- [RAIL_CancelTimer](#)(RAIL_MultiTimer_t*)
- [RAIL_IsTimerExpired](#)(RAIL_MultiTimer_t*)
- [RAIL_IsTimerRunning](#)(RAIL_MultiTimer_t*)

There are only two differences:

- Each API needs a [RAIL_MultiTimer_t](#) `timerHandle`. Every timer you plan to use should have its unique handler statically allocated.
- The callback function has a customizable, `void*` argument.

The regular RAIL timer can still be used when multi-timer is enabled. It's still implemented as a multi-timer, but its handle is allocated by RAIL. For more information on how RAIL can be used to implement multi-protocol, refer to the [Dynamic Multiprotocol User's Guide](#).

Note: multi-protocol does not necessarily mean multi PHYs. For more information on the differences between the two, refer to the [article about multi PHYs and multiprotocol](#)

Scheduling Tx

Scheduling transmissions on the RAIL time is quite simple:

```
RAIL_ScheduleTxConfig_t scheduleConfig = {
    .when = 1000,
    .mode = RAIL_TIME_DELAY,
    .txDuringRx = RAIL_SCHEDULED_TX_DURING_RX_POSTPONE_TX,
};
RAIL_StartScheduledTx(railHandle, 0, RAIL_TX_OPTIONS_DEFAULT, &scheduleConfig, NULL);
```

The parameters of [RAIL_StartScheduledTx\(\)](#) are the same as the parameters of [RAIL_StartTx\(\)](#), except the [RAIL_ScheduleTxConfig_t](#) pointer `config` argument, which works just like running the RAIL timer. The only new argument field here is `txDuringRx`; it can have the following values:

- [RAIL_SCHEDULED_TX_DURING_RX_POSTPONE_TX](#)
- [RAIL_SCHEDULED_TX_DURING_RX_ABORT_TX](#)

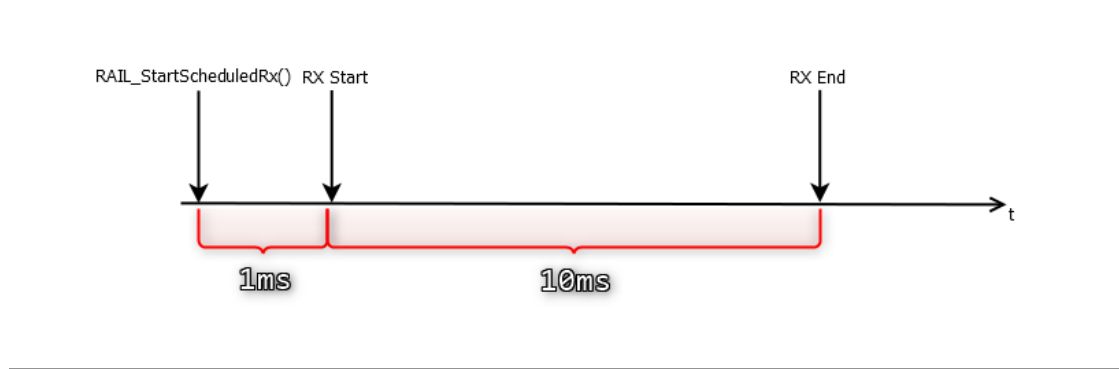
If you have the radio in receive mode, and when your scheduled packet starts transmitting, you might be in the middle of receiving a packet. In this case, `RAIL_SCHEDULED_TX_DURING_RX_POSTPONE_TX` will start the transmission when the reception receives, so transmission will be delayed from the intended scheduling. `RAIL_SCHEDULED_TX_DURING_RX_ABORT_TX` on the other hand cancels the transmission.

Scheduling Rx

Scheduling Rx is similar to scheduling Tx, but it provides more options:

```
RAIL_ScheduleRxConfig_t scheduleConfig = {
    .start = 1000,
    .startMode = RAIL_TIME_DELAY,
    .end = 10000,
    .endMode = RAIL_TIME_DELAY,
    .rxTransitionEndSchedule = false;
    .hardWindowEnd = false;
};
RAIL_ScheduleRx(railHandle, 0, RAIL_TX_OPTIONS_DEFAULT, &scheduleConfig, NULL);
```

The following illustrates the scheduled reception as defined above:



The arguments [start](#) and [startMode](#) work just like the Tx scheduling setup (or a regular timer), but for Rx, we can also configure when to end and return to idle mode. In the case above, the delay starts when the receiver starts, so the config above will start receiving 1 ms after the API call, and stop it 11 ms after the API call, closing a 10 ms Rx window.

The end timer is just an option: You can use [RAIL_TIME_DISABLED](#), in which case the radio will stay in Rx mode indefinitely. However, keep in mind that auto state transitions affect scheduled Rx mode as well; if a packet is received, it can go to idle or Tx automatically, or it can go back to Rx, in which case it will wait until the original [end](#) time, and go to idle at that point.

When the end timer fires, RAIL will trigger the [RAIL_EVENT_RX_SCHEDULED_RX_END](#) event.

The last two parameters are rarely used; see the [API documentation](#) for it.

Note also that this is one of the few APIs that behave differently in Dynamic multiprotocol (DMP) mode. For details, see [the article on DMP](#).

Multiple Radio Operations

It is not possible to schedule multiple radio operations, whether Rx or Tx. Only one operation is possible to be scheduled at a time; the next one can be scheduled when the current one is finished.

Although Dynamic multiprotocol does essentially allow it, it is, in reality rarely needed. For more info, see the [introductory article on DMP](#).

State Timing

RAIL lets you configure the delays needed for each radio state transitions described below. This is useful, in effect, in cases expecting ACKs to be received after a determined time following a transmission. The radio transceiver doesn't need to be turned on immediately, hence, saving power.

RAIL defines a few configurable transition times in [RAIL_StateTiming_t](#):

- [idleToRx](#): delay from scheduled time of Rx to actual Rx
- [idleToTx](#): delay from scheduled time of Tx to actual Tx
- [TxToRx](#): delay from Tx to Rx in auto state transition
- [RxToTx](#): delay from Rx to Tx in auto state transition

All of these are configurable with [RAIL_SetStateTiming\(\)](#). By default, all of them are set to a delay the chip can guarantee. This is somewhere between 100 μ s and 150 μ s, depending on the part number.

You can set it to 0, which is best effort. It is the fastest possible though not guaranteed switch time. It is however consistent within the same major part number (e.g., all EFR32xG23), rail version and radio config, so it can be used for tight timings.

Normal (not scheduled) Rx/Tx will also use the above configuration, but they have some additional delay because the MCU has to process the command, and might need to change the channel, which could take longer.

Calling [RAIL_StartTx\(\)](#) while the radio is in receive mode is not an Rx to Tx transition: It will first go to idle, then Tx.

There are two other delays that are not configurable:

- anything to idle
- Rx to Rx (i.e., return to Rx on the same channel, after receiving a packet)

Anything to idle usually takes about 20 μ s, while Rx to Rx is about 10 μ s, these are always best effort delays.

Search Timeout

You can change two search timeouts as well with [RAIL_SetStateTiming\(\)](#):

- [rxSearchTimeout](#) defines the length of time the radio will search for a packet when coming from idle.
- [txToRxSearchTimeout](#) defines length of time the radio will search for a packet when coming from Tx.

Both of these means that Rx will turn off automatically if the radio does not receive a sync word until the timeout. The timeout will trigger a [RAIL_EVENT_RX_TIMEOUT](#) event. This is especially useful in the second case, e.g., you can set up to wait 5 ms for acknowledgment. If you combine it with [RAIL_SetStateTiming\(\)](#), you can configure a 5 ms Rx window 1 ms after transmitted packets.

Energy Modes and Sleep

By default, RAIL timers only work in EM0 and EM1. When configured correctly, RAIL can use an RTC, which enables the timers to work almost seamlessly in EM2.

The hardware provides very accurate synchronization with RTC. Since RTC runs much slower than the radio timer (1-32 kHz vs 1 MHz), this is not as simple as saving the value of both timers at the same time: It saves the timer's value when RTC changes its value, eliminating errors caused by the synchronization.

For details, see the [article about timer synchronization](#).

API Introduced in this Article

Functions

- [RAIL_GetTime\(\)](#)
- [RAIL_SetTime\(\)](#)
- [RAIL_GetTxPacketDetails\(\)](#)
- [RAIL_GetTxPacketDetailsAlt\(\)](#)
- [RAIL_GetTxPacketDetailsAlt2\(\)](#)
- [RAIL_GetTxTimeFrameEnd\(\)](#)
- [RAIL_GetTxTimePreambleStart\(\)](#)
- [RAIL_GetTxTimeSyncWordEnd\(\)](#)
- [RAIL_GetTxTimeFrameEndAlt\(\)](#)
- [RAIL_GetTxTimePreambleStartAlt\(\)](#)
- [RAIL_GetTxTimeSyncWordEndAlt\(\)](#)
- [RAIL_GetRxPacketDetails\(\)](#)
- [RAIL_GetRxPacketDetailsAlt\(\)](#)
- [RAIL_GetRxTimeFrameEnd\(\)](#)
- [RAIL_GetRxTimePreambleStart\(\)](#)
- [RAIL_GetRxTimeSyncWordEnd\(\)](#)
- [RAIL_GetRxTimeFrameEndAlt\(\)](#)
- [RAIL_GetRxTimePreambleStartAlt\(\)](#)
- [RAIL_GetRxTimeSyncWordEndAlt\(\)](#)
- [RAIL_SetTimer\(\)](#)
- [RAIL_GetTimer\(\)](#)
- [RAIL_CancelTimer\(\)](#)
- [RAIL_IsTimerExpired\(\)](#)
- [RAIL_IsTimerRunning\(\)](#)
- [RAIL_ConfigMultiTimer\(\)](#)
- [RAIL_SetMultiTimer\(\)](#)
- [RAIL_GetMultiTimer\(\)](#)

- [RAIL_CancelMultiTimer\(\)](#)
- [RAIL_IsMultiTimerRunning\(\)](#)
- [RAIL_IsMultiTimerExpired\(\)](#)
- [RAIL_StartScheduledTx\(\)](#)
- [RAIL_ScheduleRx\(\)](#)
- [RAIL_SetStateTiming\(\)](#)

Types and Enums

- [RAIL_Time_t](#)
- [RAIL_TxPacketDetails_t](#)
- [RAIL_RxPacketDetails_t](#)
- [RAIL_PacketTimePosition_t](#) ([RAIL_PACKET_TIME_AT_PREAMBLE_START](#) / [RAIL_PACKET_TIME_AT_SYNC_END](#) / [RAIL_PACKET_TIME_AT_PACKET_END](#))
- [RAIL_TimeMode_t](#) ([RAIL_TIME_DELAY](#) / [RAIL_TIME_ABSOLUTE](#) / [RAIL_TIME_DISABLED](#))
- [RAIL_TimerCallback_t](#)
- [RAIL_MultiTimer_t](#)
- [RAIL_MultiTimerCallback_t](#)
- [RAIL_ScheduleTxConfig_t](#)
- [RAIL_ScheduleRxConfig_t](#)
- [RAIL_EVENT_RX_SCHEDULED_RX_END](#)
- [RAIL_EVENT_RX_TIMEOUT](#)
- [RAIL_StateTiming_t](#)
- [RAIL_ScheduledTxDuringRx_t](#) ([RAIL_SCHEDULED_TX_DURING_RX_POSTPONE_TX](#) / [RAIL_SCHEDULED_TX_DURING_RX_ABORT_TX](#))

Debugging

Debugging

EFR32 Wireless Geckos, like any modern MCU, supports on-chip debugging. However, when debugging radio problems, on-chip debugging is often inadequate. Generally, the problem is that some bugs originate from the real time nature of the radio. When stopping the radio at a breakpoint, you change the behavior of the target. Also, you might need information from multiple targets at the same time. In this article, we recommend RAIL/EFR32 debug techniques for these problems as well as other RAIL-specific debug features.

RAILtest

While the original goal for RAILtest was to test the RAIL library, it can be a useful tool for software development as well. In most cases, you probably want to concentrate on either receiving or transmitting, and need something to play the role of the other side. RAILtest can be simply used for that. Just set up the desired config, and it will immediately start in receive mode, printing the received messages with timestamps (useful for checking synchronized transmits). It doesn't just print; it has a command line interface to change almost anything in RAIL. Some of the most useful commands for testing:

- `rx 0` turns off the radio, while `rx 1` reenables it in receive mode
- `tx <N>` sends N message or if N=0, it will send messages until you call `tx 0` again
- `settxdelay <N>` changes the delay between transmitted packets
- `setchannel <N>` changes the channel to N
- `setconfigindex <N>` will change the radio config to the given number in a multi-PHY setup
- `status` prints a statistics (e.g., number of received packets) since the device booted
- `resetcounters` zeroes out the statistics for the `status` command
- `setrxoption 3` can be used to disable CRC check while `setrxoptions 0` reenables it
- `setfixedlength <N>` sets the configured packet length to N, for both Tx and Rx, even if the radio config is variable length
- `settxpayload` can change the packet RAILtest transmits. The first parameter is the offset and all following parameters will be written continuously
- `settxlength <N>` changes how many bytes will be written into the Tx FIFO before transmission (note that it does not actually change the transmitted length)
- `settxtransitions` and `setrxtransitions` can be used to configure automatic state transitions. Do disable going back to Rx, call it both with the argument `i i` (i as idle)
- `settxtone 1` transmits an unmodulated carrier on the channel until `settxtone 0` is called
- `getrssi` prints out the current RSSI measured on the antenna
- `startavgRSSI <N>` measures RSSI for N μ s
- `settxstream` can be used to transmit various periodic modulated signal, including PN9 pseudorandom and '01' pattern
- `setgpiooutpin` can be used to drive a pin low or high, e.g., to drive an external PA. For example, `setgpioinoutpin C 3 0` will drive portC3 down
- `setdebugsignal` outputs an internal debug signal to GPIO, this is discussed in detail below
- `reset` resets the part

For further details, refer to [RAILtest User's Guide](#).

Debugging with SWD

EFR32 provides SWD as the debug interface. You can use breakpoints and read register values when the MCU is halted, just like any other EFM32. However, you should be careful when using this tool for debugging the radio.

Keep in mind that the radio is fast; if you put a breakpoint to the RAIL event handler, you can stop the MCU while it's receiving/transmitting a packet, which could result in bugs that normally wouldn't happen, e.g., the timestamps could be completely wrong, or you can run into buffer overflow/buffer underflow events.

You could also see some strange phenomena when debugging the radio; that is, you might see that the values stored in the memory have changed, even though the MCU is halted. That is caused by the radio co-processor of EFR32 (sometimes called sequencer), which is used to help with time-critical event handling of the radio, and shares some memory with the main MCU.

Generally, we would recommend not putting breakpoints into the RAIL event handler (of course, it is fine to do it if you don't care about execution after the breakpoint); set temporary flags instead and stop the application in the main loop when it's safe to check the values. Another good technique is to put past events into a queue:

```
volatile RAIL_Events_t eventQueue[5];

void sl_rail_util_on_event(RAIL_Handle_t railHandle, RAIL_Events_t events)
{
    // Handle events
    for(int i=1; i<5; i++){
        eventQueue[i-1] = eventQueue[i];
    }
    eventQueue[4] = events;
}
```

This helps a lot to figure out the flow that resulted in the problem you're debugging.

Assertion

RAIL provides an assertion mechanism. For example, if you try to set a channel on a frequency that is not available from the radio configuration, you'll get a `RAIL_ASSERT_FAILED_SYNTH_VCO_FREQUENCY` assert. If `RAIL Utility, Callbacks` component is installed, these asserts will be printed to stdout by default. You can catch these asserts by overriding `sl_rail_util_on_assert_failed` weak function.

It's a good idea to stop normal operation on assert - in a production application, you probably want to reset from it. If you put a breakpoint in `sl_rail_util_on_assert_failed`, you can also see the callstack of the assert, where you might not understand all levels (as part of it is probably in the closed source library), but at least you can see the original source of the assert.

Note that if nothing implements `sl_rail_util_on_assert_failed`, it defaults to the rail-lib implementation, which is an infinite loop. Unless you have a watchdog, this can be harmful in production.

Debugging with GPIOs and PRS Channels

You probably already used this technique to debug time-critical problems: toggle a GPIO when something happens, and use a logic analyzer or an oscilloscope to analyze it. EFR32 can help further with radio specific PRS (Peripheral Reflex System) channels that are wired to GPIOs. If a GPIO is configured for a given PRS signal, it will go high when the signal is active; i.e., if you configure `PRS_RAC_TX` to a GPIO, the signal will be high during transmission. You can find a list of the available radio PRS signals below.

PRS signals can be extremely useful when working on multiple devices:

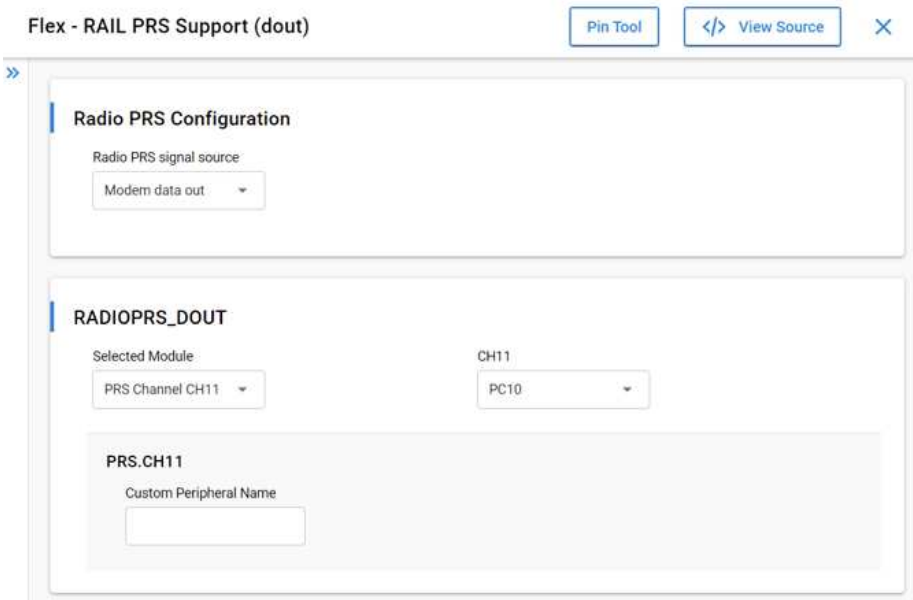


In the logic analyzer capture above (of a PRS-modified duty cycle example), you can see that device 2 starts transmitting, and how device 1 locks on that signal.

PRS signals also can be used to measure state transition times, which is often important in time-critical applications. Note, however, that even though the `RXSTATE` signal is driven high, the radio and RAIL could still need some time to complete the state transition, so it might be possible that you can't detect a signal at the beginning of the Rx window, or you're already leaving Rx mode when an event is triggered, making it useless. The magnitude of this error however is small only a few μ s.

Setting up PRS Signals with PRS Debug Signal Support Component

Simplicity Studio 5's **PRS Debug Signal Support** (in earlier SDKs, **Flex - RAIL PRS Support**) component offers an easy way to set up the radio PRS signals. After installing the component, you can create multiple component instances. Each instance corresponds to a PRS signal where you can select the PRS signal source and the output pin. In the picture below, we created an instance named `dout` and we set Modem data out to pin PC10.



Note that while this article is intended for proprietary developers, the above component and the radio PRS debug signal system is available with any stack. The only possible caveat is that some PRS channels might be used with certain stacks. E.g., PRS channel 7 is usually used for [RAIL timer synchronization](#).

Setting up PRS Signals Without PRS Debug Signal Support Component

It can be also useful to set up the PRS signals directly from the source code. The implementation is different for the device generations.

For Series 1:

```
// Useful Pins that work on most boards
// Pin, Ch, Loc, Board gen  (conflict) WSTK_P?  EXP_HEADER?
// PC10, 9, 15 Series1  (I2C)  WSTK_P12  EXP_HEADER15
// PC11, 10, 5 Series1  (I2C)  WSTK_P13  EXP_HEADER16
// PF02, 0, 2 xG1/xG13/xG14 (SWO)  WSTK_P28
// PF03, 1, 2 xG1/xG13/xG14 (TDI)  WSTK_P10  EXP_HEADER13
// PF04, 2, 2 xG1/xG13/xG14 (LED0)  WSTK_P8  EXP_HEADER11
// PF05, 3, 2 xG1/xG13/xG14 (LED1)  WSTK_P32
// PF06, 0, 6 xG1/xG13/xG14 (BTN0)  WSTK_P34
// PF07, 1, 6 xG1/xG13/xG14 (BTN1)  WSTK_P36
// PC09, 11, 2 xG12          WSTK_P10  EXP_HEADER13
// PD09, 3, 8 xG12          WSTK_P02  EXP_HEADER05
// PD10, 4, 1 xG12          WSTK_P04  EXP_HEADER07
// PD11, 5, 1 xG12          WSTK_P06  EXP_HEADER09
// PD12, 6, 14 xG12         WSTK_P08  EXP_HEADER11

// Set PRS_RAC_TX to PC10
// EXP15: PC10, Ch9, Loc15
CMU_ClockEnable(cmuClock_PRS, true);
CMU_ClockEnable(cmuClock_GPIO, true);
GPIO_PinModeSet(gpioPortC, 10, gpioModePushPull, 0);
PRS_ConnectSignal(9, prsTypeAsync, PRS_RAC_TX);
PRS_GpioOutputLocation(9, 15);
```

For Series 2:

```
// PRS Channel 0-5 can be routed to port A/B and Channel 6-11 to port C/D
// Set PRS_RACL_TX to PC04
CMU_ClockEnable(cmuClock_PRS, true);
CMU_ClockEnable(cmuClock_GPIO, true);
PRS_ConnectSignal(8, prsTypeAsync, PRS_RACL_TX);
PRS_PinOutput(8, prsTypeAsync, gpioPortC, 4);
GPIO_PinModeSet(gpioPortC, 4, gpioModePushPull, 0);
```

Setting up PRS Signals with RAILtest

In RAILtest, you can set up the PRS signals using the `setDebugSignal` serial command. For example, to route `TXACTIVE` to pin PC11 use:

```
setDebugSignal PC11 TXACTIVE
```

You can use `setDebugSignal help me` to list available GPIOs and signals.

Radio PRS Signals

For Series1:

Signal	RAILtest name	Summary
PRS_RAC_ACTIVE	RACACTIVE	Radio enabled
PRS_RAC_TX	TXACTIVE	Transmit mode enabled
PRS_RAC_RX	RXACTIVE	Receive mode enabled
PRS_RAC_LNAEN	LNAEN	LNA enabled for RX
PRS_RAC_PAEN	PAEN	PA enabled for TX
PRS_MODEM_FRAMEDET	FRAMEDETECT	Frame detected (Syncword received)

Signal	RAILtest name	Summary
PRS_MODEM_PREDET	PREAMBLEDETECT	Preamble detected
PRS_MODEM_TIMDET	TIMINGDETECT	Timing detected
PRS_MODEM_FRAMESENT	FRAMESENT	Frame sent
PRS_MODEM_SYNCSENT	SYNCSENT	Syncword sent
PRS_MODEM_PRESENT	N/A	Preamble sent
PRS_MODEM_PRESENT+1	CUSTOM_PRS 0x56 0x6	Modem clock out
PRS_MODEM_PRESENT+2	CUSTOM_PRS 0x56 0x7	Modem data out

For Series2:

Signal	RAILtest name	Summary
PRS_RACL_ACTIVE	RACTIVE	Radio enabled
PRS_RACL_TX	TXACTIVE	Transmit mode enabled
PRS_RACL_RX	RXACTIVE	Receive mode enabled
PRS_RACL_LNAEN	LNAEN	LNA enabled for Rx
PRS_RACL_PAEN	PAEN	PA enabled for Tx
PRS_MODEML_FRAMEDET	FRAMEDETECT	Frame detected (Syncword received)
PRS_MODEM_PREDET	PREAMBLEDETECT	Preamble detected
PRS_MODEMH_TIMDET	TIMINGDETECT	Timing detected (not valid for most PHYs)
PRS_MODEM_FRAMESENT	FRAMESENT	Frame sent
PRS_MODEMH_SYNCSENT	SYNCSENT	Syncword sent
OFDMFRAMEDETECT_PRSMUXLSB	OFDMFRAMEDETECT	OFDM Frame detected (Syncword received)
OFDMEOF_PRSMUXLSB	OFDMEOF	OFDM End of Frame
OFDMFRAMESENT_PRSMUXLSB	OFDMFRAMESENT	OFDM Frame sent
OFDMSYNCSENT_PRSMUXLSB	OFDMSYNCSENT	OFDM Syncword sent
OFDMSYMBOLCLK_PRSMUXLSB	OFDMSYMBOLCLK	OFDM Symbol clock out
PRS_MODEMH_PRESENT	N/A	Preamble sent
PRS_MODEML_DCLK	CUSTOM_PRS 0x56 0x5	Modem clock out
PRS_MODEML_DOUT	CUSTOM_PRS 0x56 0x6	Modem data out

The last two signals are direct outputs of the modem (just before modulation for Tx and after demodulation for RX). On Series 1 devices, they don't have proper names, so the recommended way to access them is to increment `PRS_MODEM_PRESENT` . If you are working with RAILtest, they can be accessed as custom PRS. In this case, instead of `0x56` , use:

- `0x26` on xG1,
- `0x56` on xG12-xG14,
- `0x2B` on xG21,
- `0x2A` on xG22,
- `0x29` on xG23 and newer

Note that especially on Series 2 devices, some debug signals might not work as expected, or even meaningless. `PRS_MODEML_DOUT` for example behaves differently on rx depending on the radio config while `PRS_MODEMH_TIMDET` also depends on the PHY but in most cases its behavior is undefined. For software debugging however, these signals are rarely needed.

Calibrations

Calibrations

To reach the maximum performance of the radio, you should enable and implement calibrations. In this document, we will talk about the details on how to enable and implement calibrations, when they can be avoided, and in what situations should they be implemented.

Enable All Calibrations

Generally, you should enable all calibrations. They do have some drawbacks, and they can be avoided in some cases, but unless you're sure you don't need them, it's best to enable all of them.

If you open an example in Simplicity Studio, the RAIL initialization components will enable calibrations partially. However, the application still needs to implement the calibration callback:

```
if( events & RAIL_EVENT_CAL_NEEDED ){  
    RAIL_Calibrate(rail_handle, NULL, RAIL_CAL_ALL_PENDING);  
}
```

See the [Event Handling and Automatic State Transitions](#) tutorial if you're unsure how and where to use this code.

If you want to change this default "enable everything behavior", you can change them in the components **RAIL Utility, Initialization** (for image rejection and temperature calibration) and **RAIL Utility, Power Amplifiers** (for PA calibration). Alternatively, you can also call [RAIL_EnablePaCal\(\)](#) and/or [RAIL_ConfigCal\(\)](#) in your application's init code.

In the following sections, we'll look into the 3 calibrations available on EFR32.

PA Calibration

Why Is It Needed?

The Power Amplifier (PA) has some part-to-part variations at low-power levels. To compensate this, Silicon Labs measures the PA during manufacturing and stores a few calibration constants on the DI page.

On Series 2 devices, PA calibration is enabled by default. On some devices, the API can be used to disable it, and on others, like on EFR32xG23, this API has no effect, and only exists for compatibility reasons.

On Series 1 devices, the 2.4 GHz Low Power PA is not affected by the PA calibrations.

When to Perform This Calibration?

Call [RAIL_EnablePaCal\(\)](#) before calling [RAIL_ConfigTxPower\(\)](#).

How Long Does It Take?

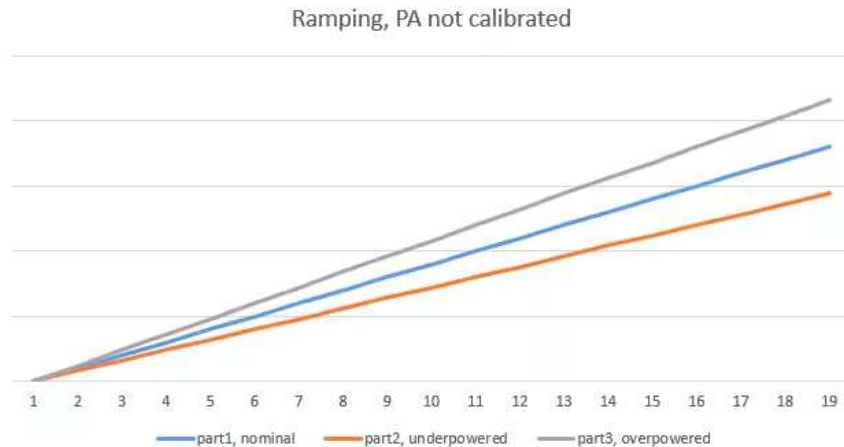
Virtually no time. It's just loading some constants from the DI page, not an actual calibration.

What if I Don't Use It?

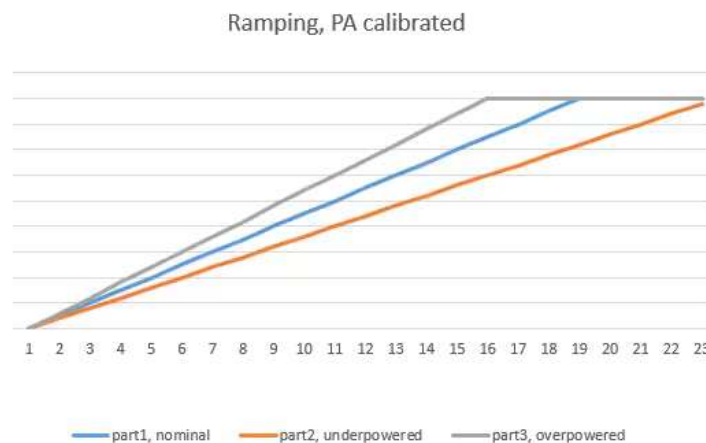
When you set to transmit at low power, your output power might be different from part-to-part.

Are There Any Drawbacks?

By default, during PA ramp-up, the radio ramps the PA current to a fixed value, which always takes the same time (but results to different power output).



With calibration, the slope will be the same, but it can reach the target output power either faster or slower. To guarantee the scheduling timing, RAIL will set up the slowest ramp time as a delay for the ramp-up. This means that ramping will take slightly longer, and it might reach the top early.



Since we're talking about microseconds here, it's usually not an issue, unless for some reason, you need to be aware of ramping shape and time.

How to Do It?

The simplest way is the implementation above.

Temperature Calibration

Why Is It Needed?

The Voltage Controlled Oscillator (VCO) must be temperature-compensated to generate the expected frequency. This will be performed automatically when the radio enters either the Tx or Rx state from idle, but it might be needed when the radio is in Rx for a long duration with big temperature changes.

When to Perform This Calibration?

Generally speaking, when RAIL requests it through [RAIL_EVENT_CAL_NEEDED](#), RAIL will request calibration when the temperature changes by 70 °C.

How Long Does It Take?

Under 100 µs.

What if I Don't Use It?

There's no need for it if your device does not stay in Rx mode (without channel change), while the temperature can change by a lot.

If you don't calibrate when requested, your receiver frequency might change to the point where reception will be impossible.

Are There Any Drawbacks?

It's not possible to receive packets while the calibration is running.

How to Do It?

Call [RAIL_Calibrate\(\)](#) or [RAIL_CalibrateTemp\(\)](#) from the [RAIL_EVENT_CAL_NEEDED](#) event if [RAIL_GetPendingCal\(\)](#) returns [RAIL_CAL_TEMP](#).

Image Rejection Calibration

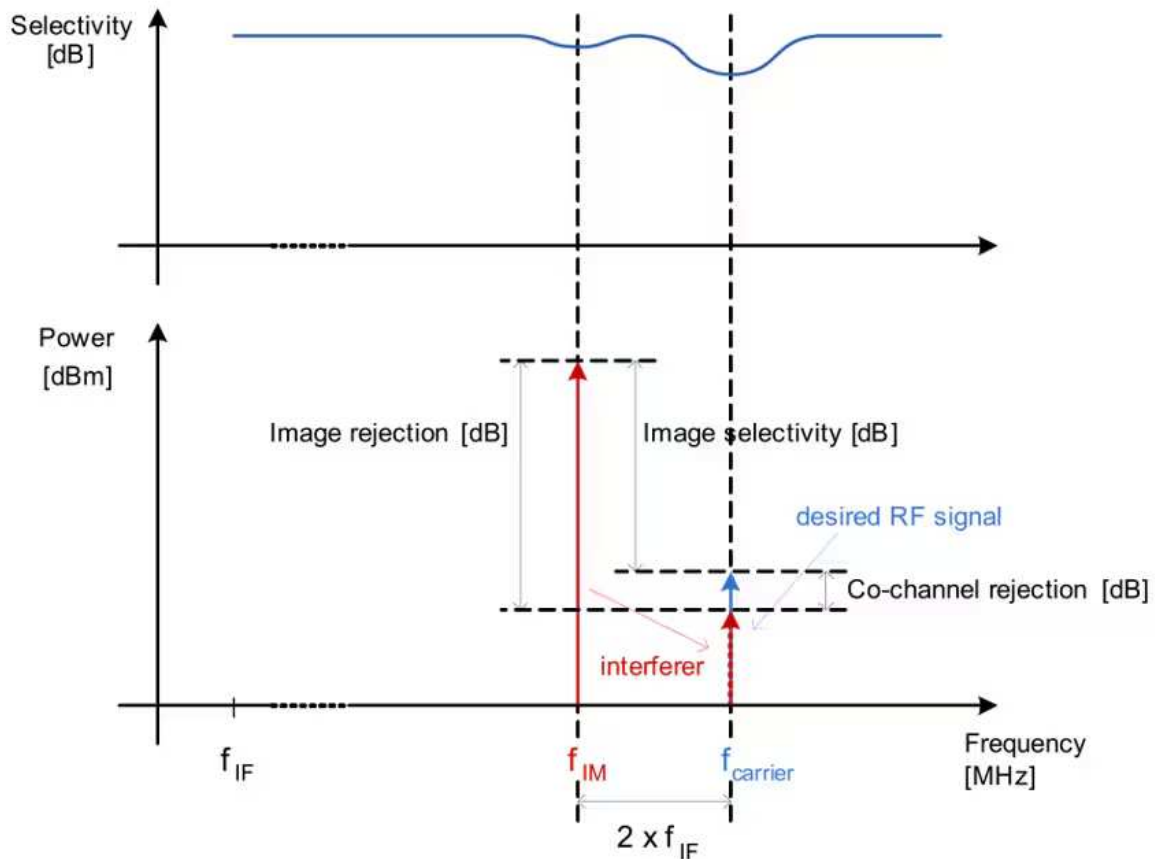
Also known as IR or IQ calibration.

Why Is It Needed?

EFR32 implements a superheterodyne receiver, which fundamentally receives signals from two frequency bands: one IF (Intermediate Frequency) above the Local Oscillator (LO) frequency - wanted band, and one IF below the LO frequency

- image band (the wanted band can be programmed to be below the LO, in which case image band is above the LO). Image band reception is not desired. It could cause reception issues if there's something on the image band, e.g. another channel or some completely different RF transmission.

The receiver is designed in a way that there is inherently suppression on the image band. The suppression of the image however can be further improved by performing an on-chip calibration. Note that the calibration does not improve receiver sensitivity, and it's not even used for Tx; it only improves signal suppression on the image frequency in Rx mode.



When to Perform This Calibration?

Once, for each subGHz radio configuration, on each RF path, in the lifetime of the chip. This means it's enough to calibrate during manufacturing, store the result, and apply it later - but it has to be done on every part. In fact, we do this measurement for 2.4 GHz 802.15.4 and BLE and store it in the DI page of the chip. The internal calibration process is not available on 2.4 GHz, but the stored values are quite good for any radio configuration on that band. However, we can't provide calibration values for all the possible sub-GHz radio configurations.

How Long Does It Take?

The calibration itself takes about 200 ms. Loading a stored value is just a register write, so it's nearly instantaneous.

What if I Don't Use It?

You will be able to receive from the image frequency easier (i.e., sensitivity on the image band will be better, but very likely still not as good as on the wanted band). This can be harmful, for example if the image frequency is a neighboring channel.

Are There Any Drawbacks?

Loading a stored value has no drawback. The calibration itself takes quite a long time, and it needs about 3 kB code memory (in Flex 3.0.2, with GCC 7.2.1, on Series 1). On EFR32xG23 and newer parts, it should need less code memory, as part of the algorithm is implemented in the hardware. However, it still needs about 2.3 kB (in Flex 4.3.1, with GCC 10.3.1).

How to Do It?

The easiest way is the automatic implementation above. RAIL will request calibration when you call `RAIL_ConfigChannels()`. However, you can also trigger calibration manually, e.g., at boot.

There are a few more details that are important in multi-phy or multiprotocol environments. We will discuss it in [the multiply article](#).

IR calibration requires the radio Tx looped back to its Rx, and this cannot be easily done in some configurations. If all of these are true for you, please contact our support team on how to implement IR calibrations:

- you have a Series 1 device (such as EFR32xG1/xG12/xG13/xG14)
- the matching has split Tx/Rx configuration (practically in case of using FEM)
- the target frequency is above 462 MHz

Related Variables in the Radio Configuration

IR Calibration is configured by a variable from `rail_config.c`, and the end result is cached in a variable of `rail_config.c`. For details, see the [article on the rail_config.c file](#), specifically the section *IR Calibration helpers*.

Using the Alternate APIs

If you can use a "manufacturing test" firmware to run the IR calibration and store it, and only load the value from the end-product application, you can remove the significant code space required by the IR calibration by using the alternate API:

```
if( events & RAIL_EVENT_CAL_NEEDED ){
    RAIL_CalMask_t pending = RAIL_GetPendingCal(rail_handle);
    if ( pending & RAIL_CAL_ONETIME_IRCAL ){
        RAIL_ApplyIrCalibration(rail_handle, storedIrValue);
    } else { //if not IR, it must be temperature on EFR
        RAIL_CalibrateTemp(rail_handle);
    }
}
```

In the manufacturing firmware, you can use `RAIL_CalibrateIr()` to get the IR calibration value. If you never call `RAIL_Calibrate()` or `RAIL_CalibrateIr()` in your application, the calibration code will be removed freeing up code space.

IR Calibrations with Multiple RF Paths

Some Series 2 devices (e.g., EFR32xG23) have an internal RF switch to support two RF paths.

With these parts, the above recommended APIs calibrate the antenna which was selected via `RAIL_ConfigAntenna()`.

Antenna Diversity

If you use the RF switch for antenna diversity, it is recommended to use `RAIL_CalibrateIrAlt()` instead, and calibrate for both antennas, one by one:

```
RAIL_CalibrateIrAlt(rail_handle, NULL, RAIL_ANTENNA_0);
RAIL_CalibrateIrAlt(rail_handle, NULL, RAIL_ANTENNA_1);
```

Using `RAIL_CalibrateIrAlt()` is however not recommended on parts with internal switch, if you do not want to use antenna diversity. The non-Alt API always calibrates the antenna configured for use, so it is less error-prone.

Different Matching

Another common usecase is to have multiple matching circuits (e.g., one for 433 MHz and another for 915 MHz) on the same device, and maybe also with an external switch to use the same antenna. In practice, you will use the two paths with different PHYs.

In this case, it is recommended to rely on RAIL's storage of calibration values for multi-PHY configs (see the [Multi-PHY article](#) for more details). However, you have to be careful to select the right matching and antenna for the given PHY before performing the calibration, and it is recommended to use `RAIL_Calibrate()` or `RAIL_CalibrateIr()`, not `RAIL_CalibrateIrAlt()`.

Stack-Specific IR Calibration

Bluetooth Low-Energy and OpenThread

Since BLE and OpenThread operate on the 2.4 GHz band, as it was described above, IR calibration is performed by applying the calibration value from the DI page.

Zigbee

In case of the Zigbee, the IR calibration is performed at one time, once the radio starts using a band (2.4 GHz, 863 MHz or 915 MHz) first time from the last boot.

On 2.4 GHz band it is a simple memory write. On sub-GHz bands the IQ calibration differs on different device generations:

- EFR32xG1 ICs are supplied with the proper calibration stored in the DI page, similarly to the 2.4 GHz band calibration.
- All other parts run IR calibration when the device starts using these bands, and the results are retained until reboot.

Z-Wave

The calibration in the Z-Wave stack is done automatically, during PHY load, implemented in the stack.

API Introduced in This Article

Functions

- [RAIL_EnablePaCal\(\)](#)
- [RAIL_ConfigCal\(\)](#)
- [RAIL_ApplyIrCalibration\(\)](#)
- [RAIL_Calibrate\(\)](#)
- [RAIL_CalibrateTemp\(\)](#)
- [RAIL_CalibrateIr\(\)](#)
- [RAIL_CalibrateIrAlt\(\)](#)
- [RAIL_GetPendingCal\(\)](#)

Types and Enums

- [RAIL_CalMask_t](#)
- [RAIL_CAL_ALL](#)
- [RAIL_CAL_ALL_PENDING](#)
- [RAIL_CAL_ONETIME_IRCAL](#)
- [RAIL_CAL_TEMP](#)
- [RAIL_EVENT_CAL_NEEDED](#)

Power Manager

Power Manager Integration

In this article, we describe what are the supported RAIL features in different energy modes, and how high energy efficiency can be achieved with appropriate software. Furthermore, we give a general overview of how RAIL leverages the Power Manager's services.

Introduction

The radio must always run off of a High-Frequency Crystal Oscillator (HFXO) that drives the RAIL's internal timer for high accuracy protocol timing. It is essential to keep this clock running during any active radio operation such as active transmission and reception. As a result, the radio is active only in EM0 and EM1/EM1P as the HFXO clock is running only in these - highest - Energy Modes (EM).

The Power Manager is a platform-level software module that manages the system's energy modes. Its primary purpose is to transition the system to the lowest-energy mode, consequently reducing the overall current consumption, when the processor has nothing to execute and there is no peripheral requesting higher energy modes. Additionally, it ensures strict control of the power-intensive resources, just like the high-frequency external oscillator.

You can instruct the Power Manager to enable/disable energy mode transitions by specifying EM level requirements. You can also register listeners at any point of your software to get notification on the energy mode transitions via the Power Manager's notification mechanism.

For complete documentation on the Power Manager service, visit [the platform documentation page](#).

Best Practices for using RAIL with Power Manager

Benefits of Using Power Manager

The Power Manager makes your application energy-friendly more effectively and it provides a neat API to interface with. As some of the Silicon Labs software modules might tightly integrate Power Manager (e.g., the [Sleep Timer](#) service and RAIL), your application's energy mode requirements can be managed in the same way as those do. That makes complex decisions on the energy mode requirements of the entire software much more coherent.

The Power Manager offers a notification system that keeps the "subscribers" informed on energy mode changes providing the perfect opportunity to react to the EM transitions as soon as possible.

Enabling Power Manager and Timer Synchronization in RAIL

The [RAIL_InitPowerManager\(\)](#) configures RAIL to use Power Manager. If it is initialized, RAIL automatically request the lowest energy mode sufficient for proper radio operation. To stop this feature, call [RAIL_DeinitPowerManager\(\)](#). However, removing the energy mode requirements is not done automatically as it is discussed in the following section.

If you have to schedule radio operations more precisely during the sleep periods, you also need to call the [RAIL_ConfigSleep\(\)](#) or [RAIL_ConfigSleepAlt\(\)](#) APIs with the `RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED` parameter. If the synchronization is not needed, use the `RAIL_SLEEP_CONFIG_TIMERSYNC_DISABLED` option. Details on how this synchronization works are [documented in a later article](#).

Installing the **Power Manager Integration** component under **RAIL-SDK** configures RAIL to use Power Manager as well as enables timer synchronization.

Radio Yielding Practice

When RAIL's Power Manager integration is enabled, any operation that requires the radio to be in Rx or Tx will request the power mode needed for that, usually EM1. [RAIL_Idle\(\)](#), which stops any of these operations, will release that power mode request.

In case the radio returns to idle automatically however, e.g., at Tx complete, the application must explicitly release the power mode by calling [RAIL_Yield\(\)](#) or [RAIL_Idle\(\)](#).

Timers also require EM1, at least before their callback is called, so calling [RAIL_Yield\(\)](#) (or [Idle](#)) from a timer callback is also necessary.

To summarize, you will need to call [RAIL_Yield\(\)](#) or [RAIL_Idle\(\)](#) from the rail timer callbacks and event handler to make sure RAIL releases the requested power mode. In the event handler, Yielding is mainly necessary at Tx completion and Rx completion, if automatic state transitions are set to idle. There are other events however that reports that radio or timer is not needed anymore, such as `RAIL_EVENT_RSSI_AVERAGE_DONE` and `RAIL_EVENT_TX_SCHEDULED_TX_MISSED`.

Note that this could mean that you have to enable some events just to yield the requested energy mode, typically the events in the groups `RAIL_EVENTS_RX_COMPLETION` and `RAIL_EVENTS_TX_COMPLETION`.

EM4 Limitation

Because waking up from the lowest - EM4 - energy mode resets the MCU entirely, this scenario does not fit into how the Power Manager operates. Therefore entering and leaving the lowest energy mode is not supported by the Power Manager. Still, this mode can be requested in the application as well as restoring previous states is also the application's duty. Keep in mind that entering EM4 aborts all ongoing and scheduled radio operations and makes time synchronization impossible.

RAIL without Power Manager

If you've installed the Power Manager service in your application, but it is not initialized to be used by RAIL (or the Power Manager is temporarily deinitialized), you should manually manage the energy mode requirements matching the radio's needs.

As discussed above, any active radio operation requires EM1(P) or EM0 energy mode. If RAIL does not initialize Power Manager, call `sl_power_manager_add_em_requirement()` and `sl_power_manager_remove_em_requirement()` with the required energy mode before starting the radio operation and after the associated radio event has been handled, respectively.

Synchronization requires EM2 because this is the highest energy mode with active low-frequency oscillators running. [RAIL_Sleep\(\)](#) and [RAIL_Wake\(\)](#) APIs must be called for performing time synchronization. Find more implementation details on docs.silabs.com.

Entering lower energy modes than required breaks the active radio operation or the synchronization mechanism.

API Introduced in this Article

Functions

- [RAIL_Sleep\(\)](#)
- [RAIL_Wake\(\)](#)
- [RAIL_InitPowerManager\(\)](#)
- [RAIL_DeinitPowerManager\(\)](#)
- [RAIL_ConfigSleep\(\)](#)
- [RAIL_ConfigSleepAlt\(\)](#)
- [RAIL_Yield\(\)](#)

Types and Enums

- [RAIL_SleepConfig_t](#) ([RAIL_CONFIG_TIMERSYNC_DISABLED](#) / [RAIL_CONFIG_TIMERSYNC_ENABLED](#))
- [RAIL_TimerSyncConfig_t](#)

Idle Modes

Idle Modes

The function `RAIL_Idle()` takes a mode argument, with which you select one of the four available methods of idling the radio to use. Though the [API documentation briefly describes each mode](#), it might be difficult to understand their differences without seeing examples and use cases. This document aims to help you more clearly understand each mode.

General Recommendations

In most cases, `RAIL_IDLE` is the recommended mode to use with this API. `RAIL_IDLE_ABORT` is helpful when you want to also abort an ongoing ("active") Tx or Rx operation.

On the other hand, `RAIL_IDLE_FORCE_SHUTDOWN` is not recommended for use, and `RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS` should be handled with care.

RAIL_IDLE

The mode `RAIL_IDLE` turns off the radio after the current operation is finished. It also cancels any transmit or receive scheduled in the future. Current operations that won't be aborted include:

- active transmit, which starts when the first bit (usually preamble) is transmitted and ends when the last bit (usually CRC) is transmitted.
- active receive, which starts at sync word detect, and ends when the last bit is received (which depends on the decoded length of the packet).

RAIL_IDLE_ABORT

The mode `RAIL_IDLE_ABORT` works the same as `RAIL_IDLE`, but it will also abort active operations. However, `RAIL_IDLE_ABORT` will always wait for a stable state before turning off the radio, e.g. if the radio was preparing to enter Rx mode and is waiting for the PLL to lock, RAIL will wait until the Tx state is reached before idling the radio.

RAIL_IDLE_FORCE_SHUTDOWN

Unlike `RAIL_IDLE_ABORT` (which waits for a stable radio state), `RAIL_IDLE_FORCE_SHUTDOWN` immediately forces the radio off. As this is an abrupt transition, it may corrupt data in the receive or transmit buffers. This buffer corruption can only be resolved by resetting the FIFO, which loses data, and can also consume additional time.

In our experience, it's almost always slower than `RAIL_IDLE_ABORT`, so the costs typically outweigh the benefit of using `RAIL_IDLE_FORCE_SHUTDOWN` (except when recommended by our support team for diagnostic purposes).

RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS

The mode `RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS` works the same as `RAIL_IDLE_FORCE_SHUTDOWN`, but it also clears any pending events. For more details on scenarios where this can be useful, see the [article on RAIL interrupt safety](#).

Sleep Timer Synchronization

Timer Synchronization and Sleep

Why Timer Synchronization is Needed

RAIL internal timebase provides timing used for both timestamping and scheduled transmit/receive operations. However, since the hardware timer from which the RAIL timebase is derived from is the chip's high frequency oscillator (HFXO), it is therefore not available in deep sleep mode (EM2 and lower).

Additionally, the RAIL timebase cannot be derived from a low frequency oscillator as the resolution they provide isn't sufficient. However, accuracy is only required in active transmission or reception. When the radio transceiver is inactive, which is the case in low power energy modes, low frequency clock accuracy is accurate enough. This raises the problem of high frequency/accuracy time base recovery, which is necessary for radio functions when waking up from deep sleep modes.

A mechanism allowing clock conversion from low to high frequency clock domain can be used to recover the radio transceiver timebase. It involves one of the EFR32's Real Time Counter (RTC) which is a hardware timekeeping feature that remains functional in EM2. The RAIL driver software makes sure the low and high frequency time domains are kept synchronized.

Synchronization between high and low frequency clock domain isn't a trivial problem: the typical frequency of the RTC is 32.768 kHz, which indicates a 30 μ s resolution (and it might be much worse if RTC is prescaled). If we save the value of both timers at the same time, we immediately have up to 30 μ s error, and the same error again when we restore the RAIL timebase. That can amount to 60 μ s in some cases, which is catastrophic for highly time sensitive protocols like BLE. For example, the BLE Inter-Frame Space (IFS) is 150 μ s for indication, so 60 μ s represents 40% timing error.

The solution to that problem is to save/restore the RAIL timer when the RTC timer ticks. This way, the worst case error is twice the RAIL timer resolution, which is 0.5 μ s/2 μ s, depending on the hardware (2 μ s for EFR32xG1, 0.5 μ s for all others).

Note: This refers to the hardware timer used to generate the RAIL timebase. The resolution of the RAIL timebase itself is 1 μ s for all Series 1 and 2 chip except for the EFR32xG1 family that has a resolution of 2 μ s (same as the hardware timer it relies on).

Implementation in RAIL

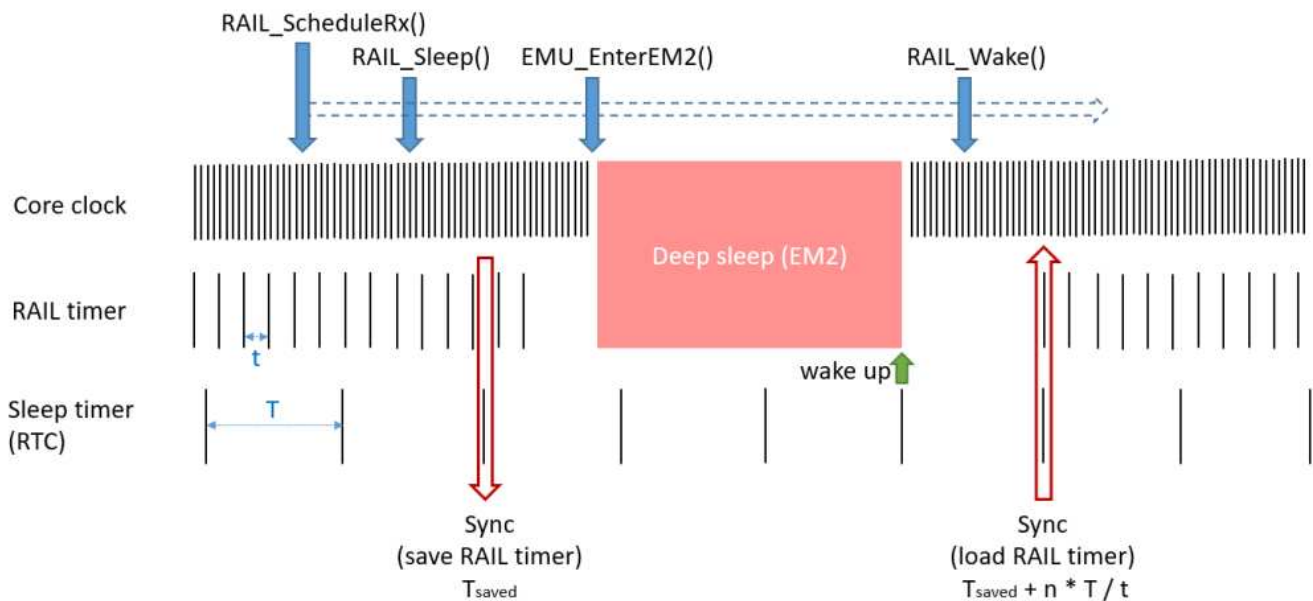
Not only RAIL synchronizes the RAIL and RTC timers before entering deep sleep and after wakeup. It also uses RTC compare to set up an interrupt, which will wake up the MCU for the next radio operation (a PRS channel is reserved for this purpose).

If RAIL has been configured to use the power manager, calling [RAIL_InitPowerManager\(\)](#) at initialization will automatically perform timer synchronization if the EM transitions is performed by the power manager. Calls to [RAIL_Sleep\(\)](#) and [RAIL_Wake\(\)](#) are unsupported in such a scenario. Power manager support in RAIL can be terminated by calling the [RAIL_DeinitPowerManager\(\)](#) routine.

The following steps indicate how the functionality works:

1. The application code start by calling [RAIL_InitPowerManager\(\)](#) and [RAIL_ConfigSleep\(\)](#) during init.
2. The application code schedules a radio operation (e.g. scheduled Tx). RAIL requests EM2 from power manager.
3. The power manager decides to enter EM2, and notifies all modules that need it, including RAIL.
4. RAIL synchronizes its hardware and the RTC timers.
5. RAIL sets up an RTC timer interrupt to trigger before the already set RAIL timer corresponding to the scheduled RX would trigger.
6. RAIL internal hardware timer is turned off.
7. The chip enters EM2 sleep.

8. RTC wakes up the MCU.
 9. The power manager restarts the main oscillator and other required clocks.
 10. The power manager notifies all modules that need it, including RAIL that the chip is returned from EM2.
 11. RAIL synchronizes the timers.
 12. RAIL timer is turned back on.
- RAIL requests EM1 from the power manager as we're close to the scheduled radio operation.



Note that the internal RAIL hardware timer is still needed to start the actual radio operation, so step 11 must be finished before the RAIL timer should be turned back on. This means RAIL must know the time it takes to perform steps 8-11, and set the RTC timer earlier by this amount, to guarantee that the RAIL timer is running by the time the radio operation must start. This time must be identified by the application if power manager is not being used.

Calling the `RAIL_ConfigSleep()` routine before the first synchronization is mandatory for timer synchronization: this function enables (`RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED`) or disables (`RAIL_SLEEP_CONFIG_TIMERSYNC_DISABLED`) timer synchronization in RAIL, depending on the `RAIL_SleepConfig_t` argument. This will be handled internally by the power manager.

It is recommended to use the power manager driven synchronization mechanism. However, if you are interested in using low energy modes without using the power manager, you can visit the old [RAIL Tutorial: Timer Synchronization and Sleep](#) article or the examples codes provided in docs.silabs.com.

Prerequisites

For the operations above, RAIL needs:

- The RTC timer should be free to use (for the synchronization).
- An RTC compare channel should be free to use (for waking up the MCU).
- A PRS channel is needed (so the synchronization can happen exactly when the RTC ticks).

The RTC timer is automatically initialized and the synchronization is managed by the `Sleep Timer` service component, while the `Power Manager` component handles the EM transitions when it is possible.

Series 2 parts have a dedicated RTC timer for this purpose.

Customizing Sleep Timer and PRS Configuration

The set of the available RTC timers varies on different chip families. You can select the appropriate RTC timer in the Sleep Timer component's configuration.

RAIL provides `RAILCb_ConfigSleepTimerSync()` callback API to configure the RTC and PRS channels. If that callback is not implemented, the default will be used, which is:

PRS channel 7

- A platform specific RTC (Usually ProRTC, if not available, compare channel 0 of RTC/RTCC)

You can also change the configuration in runtime, using the [RAIL_ConfigSleepAlt\(\)](#) API.

API Introduced in this Article

Functions

- [RAIL_Sleep\(\)](#)
- [RAIL_Wake\(\)](#)
- [RAIL_InitPowerManager\(\)](#)
- [RAIL_DeinitPowerManager\(\)](#)
- [RAIL_ConfigSleep\(\)](#)
- [RAIL_ConfigSleepAlt\(\)](#)

Types and Enums

- [RAIL_SleepConfig_t](#) ([RAIL_CONFIG_TIMERSYNC_DISABLED](#) / [RAIL_CONFIG_TIMERSYNC_ENABLED](#))
- [RAIL_TimerSyncConfig_t](#)

Tx/Rx Options

Tx/Rx Options

RAIL provides some useful options when transmitting and receiving packets. In this document, we're going to cover those. Also, the *Listen Before Talk* as well as the *Address Filtering* features will be discussed in this article.

Tx Options

The main Tx options should be passed to RAIL using the `RAIL_TxOptions_t` parameter of the `RAIL_StartTx` APIs. The available options are practically implemented as a bitmask type:

- `RAIL_TX_OPTION_WAIT_FOR_ACK` : Waits for ACK after transmitting a packet. I.e., the radio will automatically switch to Rx mode after Tx (disregarding auto state transitions) with a given timeout.
- `RAIL_TX_OPTION_REMOVE_CRC` : Ignores the config's CRC setting and does not transmit CRC.
- `RAIL_TX_OPTION_SYNC_WORD_ID` : If set, Tx will use the second configured sync word. If unset, it will use the first one.
- `RAIL_TX_OPTION_ANTENNA0` : Uses antenna 0 (usually used when 2 antennas are available for antenna diversity).
- `RAIL_TX_OPTION_ANTENNA1` : Uses antenna 1 (usually used when 2 antennas are available for antenna diversity).
- `RAIL_TX_OPTION_ALT_PREAMBLE_LEN` : Uses an alternative preamble length configured by the `RAIL_SetTxAltPreambleLength()` API.
- `RAIL_TX_OPTION_CCA_PEAK_RSSI` : Uses peak RSSI measurement in CCA instead of averaging (detailed later).
- `RAIL_TX_OPTION_CCA_ONLY` : With this option the transmit after the CCA should be started manually (detailed later).
- `RAIL_TX_OPTION_RESEND` : If the Tx FIFO has a single, complete packet in it, this option can be used to send it over and over without reloading the FIFO.
- `RAIL_TX_OPTION_CONCURRENT_PHY_ID` : Selects the Tx PHY when concurrent mode is used in the Wi-SUN stack.

If both `RAIL_TX_OPTION_ANTENNA0/1` option is not set, the active antenna will be the one, which was selected by the last Tx/Rx operation.

`RAIL_TX_OPTION_CCA_PEAK_RSSI` isn't supported on EFR32xG1 platforms.

The usual groups are also defined: `RAIL_TX_OPTIONS_ALL`, `RAIL_TX_OPTIONS_NONE` and `RAIL_TX_OPTIONS_DEFAULT` (which is the same as `RAIL_TX_OPTIONS_NONE`).

Rx Options

Rx options can be configured using `RAIL_ConfigRxOptions()`, which takes two `RAIL_RxOptions_t` parameter, `mask` and `options`. This way, it's easy to change the options individually, without tweaking the rest. The available options are:

- `RAIL_RX_OPTION_STORE_CRC` : By default, RAIL does not store the CRC, only checks it. By enabling this, it will store it as well. It can be used for example for sniffer applications.
- `RAIL_RX_OPTION_IGNORE_CRC_ERRORS` : Frames will be received even if it failed CRC check. When enabled, the result of the CRC check will be stored in the `RAIL_RxPacketDetails_t.crcPassed` field.
- `RAIL_RX_OPTION_ENABLE_DUALSYNC` : Enables the detection of both sync word.
- `RAIL_RX_OPTION_TRACK_ABORTED_FRAMES` : RAIL aborts frame reception when it filters it using the address filter. With this option enabled, it will receive the full frame (but still generates the correct error event) so the network analyzer can display it (through PTI).
- `RAIL_RX_OPTION_REMOVE_APPENDED_INFO` : Disables the storage of the appended info. Appended info is used to store the base of `RAIL_RxPacketDetails_t`, like timestamp and RSSI. If not needed, it can be disabled, saving some buffer space.
- `RAIL_RX_OPTION_ANTENNA0` : Uses antenna 0 (usually used when 2 antennas are available for antenna diversity).
- `RAIL_RX_OPTION_ANTENNA1` : Uses antenna 1 (usually used when 2 antennas are available for antenna diversity).
- `RAIL_RX_OPTION_ANTENNA_AUTO` : Uses both antennas, with antenna diversity (must be configured before using it).

- `RAIL_RX_OPTION_DISABLE_FRAME_DETECTION` : Disables the frame detection system. Useful, if you only want to check the energy of a channel, and do not want to receive frames, because it would break the application's state machines.
- `RAIL_RX_OPTION_CHANNEL_SWITCHING` : Enables concurrent listening on two IEEE 802.15.4 channels, with duty cycling. It requires a specially tuned radio config and only available on a few platforms.
- `RAIL_RX_OPTION_FAST_RX2RX` : Speeds up returning to receive mode after finished (or aborted) receiving a packet. However, it will generate an Rx error event if packet reception is completed while the previous packet is still processing.
- `RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION` : Enables collision detection required by the Wi-SUN stack.

`RAIL_RX_OPTION_IGNORE_CRC_ERRORS` affects also on ACK packets and Auto State transition: packets with CRC error will be handled as successfully received packets.

* `RAIL_RX_OPTION_ENABLE_DUALSYNC` is not available for certain combinations of chips, PHYs, and protocols. Also, it is not possible to listen only for SYNC2.

`RAIL_RX_OPTION_ANTENNA_AUTO` is not a unique value, it's actually `RAIL_RX_OPTION_ANTENNA0`/`RAIL_RX_OPTION_ANTENNA1`

The usual groups are also defined: `RAIL_RX_OPTIONS_ALL` , `RAIL_RX_OPTIONS_NONE` and `RAIL_RX_OPTIONS_DEFAULT` (which is the same as `RAIL_RX_OPTIONS_NONE`).

Some options, like the antenna selections will restart receive mode (if changed in receive mode).

Listen Before Talk Options

Listen before talk setups are often required in protocols, and a good idea to implement in some proprietary protocols as well. It solves most collisions between transmissions.

Listen before talk operations are built up from 3 tasks:

- CCA (Clear Channel Assessment): the "listen" itself, where the radio is in Rx, measuring RSSI, and decides for or against transmission
- Backoffs: randomized delays between CCA operations
- Transmission: this is the 'talk' part, in which the radio sends the preconfigured packet

Measuring RSSI means averaging during a predefined period in most cases. However, setting the `RAIL_TX_OPTION_CCA_PEAK_RSSI` option makes it available to use the peak instead of the mean RSSI value.

Generally, the radio starts with CCA: If the RSSI is below threshold, it starts transmitting. If not, it waits a random backoff period and tries CCA again. The radio should repeat this with increasing backoff periods, until successful transmission or timeout.

CCA period on EFR32 is derived from symbol time, so the minimum and maximum configurable values can differ from radio config to radio config.

The backoffs are implemented on the same timer that handles the scheduling, so it could cause problems if you start a listen before start operation from a scheduled Rx. Also, scheduled listen before talk Tx operations are not supported.

CSMA/CA

The most common listen before talk protocol is CSMA/CA (Carrier-sense Multiple Access with Collision Avoidance), which is fully supported by the EFR32 hardware. CSMA/CA defines backoffs as a random number with even distribution between `0` and `unit * (2^BE - 1)` , where `BE` is the backoff exponent, which increments with each retries.

To trigger a CSMA/CA transmission, you can start transmitting with `RAIL_StartCcaCsmatx()` instead of `RAIL_StartTx()` . The API has an extra argument, with the type of `RAIL_CsmaConfig_t`, that configures the CSMA/CA parameters:

- `csmaminboexp` and `csmamaxboexp` are setting the minimum and maximum backoff exponent (i.e., the first value of `BE` will be `csmaminboexp` , and `BE` increments up to `csmamaxboexp`).
- `csmatries` sets the number of tries (i.e., how many CCA should be performed before reporting back a busy channel).
- `ccathreshold` sets a threshold for comparing against the measured RSSI during the CCA; if the measured RSSI exceeds the threshold the channel will be judged busy.
- `ccaBackoff` is the backoff `unit` in μ s.

- `ccaDuration` sets the duration the CCA should take in μs (i.e., how long it should measure the RSSI).
- `csmaTimeout` sets the timeout in μs : if the CSMA/CA operation exceeds this timeout it will stop. This is useful, since due to the random backoffs, the CSMA/CA duration can vary. Set it to `0` to disable the timeout.

`ccaBackoff` will be truncated if it is higher than 511. Setting `csmaTries` to 0 will disable CSMA/CA: calling `RAIL_StartCcaCsmatx()` or `RAIL_StartTx()` has the same result.

There's also a special setup: if `csmaMinBoExp` and `csmaMaxBoExp` are both set to 0, that means a fixed backoff configuration. In this case, the backoff will always take the time set in `ccaBackoff`. With fixed backoff the `ccaBackoff` won't be truncated.

LBT

Another common listen before talk protocol is the obviously named LBT. It is defined in ETSI EN 300 220-1, under the Polite Spectrum Access chapter.

LBT defines backoffs as a `unit * BO`, where `BO` is a random number with even distribution between a minimum and maximum.

LBT is not supported by the EFR32 hardware, it is converted to a CSMA/CA configuration in RAIL. Because of this, if you can choose between the two, you should choose CSMA/CA.

Note that LBT can be configured with a minimum backoff limit, while CSMA/CA's minimum backoff period is always 0. This is not supported by the hardware, so LBT is implemented with a bit of cheating. The CCA period is set to the configured `period + unit * minBO`. It doesn't really matter if the CCA period is longer, and this guarantees that the minimum backoff is fulfilled.

Another limitation is that the maximum backoff must be power of 2, so the configured backoff will be rounded to the closest power of 2.

Much like CSMA/CA, LBT has its own `StartTx` API, `RAIL_StartCcaLbtTx()`, which has a familiar config struct, `RAIL_LbtConfig_t` to configure the LBT parameters:

- `lbtMinBoRand` and `lbtMaxBoRand` are setting the limits of the random backoff, just like with CSMA/CA
- `lbtTries` sets the number of tries, similarly to CSMA/CA
- `lbtBackoff` is the backoff `unit` in μs
- `lbtThreshold` is a similar threshold what we've seen in CSMA/CA
- `lbtDuration` sets the duration the CCA should take in μs
- `lbtTimeout` sets the same timeout as CSMA/CA

Setting `lbtTries` to 0 will disable LBT, just like with CSMA/CA.

Custom Listen Before Talk

A single CCA operation can be started using `RAIL_TX_OPTION_CCA_ONLY` and with `csmaTries` set to 1. With this and software timers, a custom listen before talk protocol can be implemented. Additionally, the RSSI API (`RAIL_GetRssi()` and `RAIL_StartAverageRSSI()`) can be also used for similar purposes.

Address Filtering

EFR32 supports hardware based address filtering, as long as the address is stored in the same location in all the frames (unlike 802.15.4, which is only supported through the [protocol specific API](#)). EFR32 supports **2 address fields** (e.g., for group and device address), both can be **up to 8 Bytes long**. Both fields can be configured for **up to 4 addresses** (e.g., for broadcast, unicast and 2 multicast address). Any combination can be allowed on these two fields, e.g., enable `address1` for `field0` only if `field1` has `address1`, or if `field0` has `address2`, anything is allowed on `field1`.

Enabling the address filter starts with configuring it, using:


```
RAIL_AddrConfig_t addrConfig = {
    .offsets = {2, 0},
    .sizes = {4, 0},
    .matchTable = ADDRCONFIG_MATCH_TABLE_SINGLE_FIELD,
};
RAIL_ConfigAddressFilter(railHandle, &addrConfig);
```

Its only parameter (apart from the usual `railHandle`) is `RAIL_AddrConfig_t`. `RAIL_AddrConfig_t` configures:

- `offsets[(2)]`, which sets the offset of the addresses in the frame. For the first field, from the end of sync word; for the second field, from the end of the first field. Note that this means that the fields cannot overlap.
- `sizes[(2)]`, which sets the size of the fields in byte. Setting it to 0 disables the field.
- `matchTable`, which configures the enabled combinations of the fields.

If you only use a single field, you can safely use `ADDRCONFIG_MATCH_TABLE_SINGLE_FIELD`. If you need both address fields, you should understand how the match table works. That is out of the scope of this article, but you can rely on the [API documentation](#).

Next, you load the addresses, using:

```
uint8_t address[4] = {0xAA, 0xBB, 0xCC, 0xDD};
RAIL_SetAddressFilterAddress(railHandle, 0, 1, address, true);
```

Which will enable the address 0xAABBCCDD on field 0, index 1, in enabled state. Make sure your address length matches what you configured for the field. You can later disable (and re-enable) it with:

```
RAIL_EnableAddressFilterAddress(railHandle, false, 0, 1);
```

Note that this API is referencing the address by its field and index. If you want to enable/disable addresses based on the value, i.e., the address itself, you should track which index is used for each address.

Finally, to enable (or disable) the whole address filter, use:

```
RAIL_EnableAddressFilter(railHandle, true);
```

Once address filter is enabled, you can get notifications on filtered frames using the `RAIL_EVENT_RX_ADDRESS_FILTERED` event.

API Introduced in this Article

Functions

- [RAIL_SetTxAltPreambleLength\(\)](#)
- [RAIL_ConfigRxOptions\(\)](#)
- [RAIL_StartCcaCsmaTx\(\)](#)
- [RAIL_StartCcaLbtTx\(\)](#)
- [RAIL_ConfigAddressFilter\(\)](#)
- [RAIL_SetAddressFilterAddress\(\)](#)
- [RAIL_EnableAddressFilterAddress\(\)](#)
- [RAIL_EnableAddressFilter\(\)](#)

Types and Enums

- [RAIL_TxOptions_t](#)
- [RAIL_RxOptions_t](#)
- [RAIL_CsmaConfig_t](#)
- [RAIL_LbtConfig_t](#)

- [RAIL_AddrConfig_t](#)

FIFO Mode

FIFO Mode

Like most embedded radio platforms, EFR32 provides a way to download/upload packets during reception/transmission, respectively. So, for example, while you're sending the first couple of bytes of your frame, you can still upload the end of the frame.

Since the radio is using ring buffers, this makes it possible to handle longer frames than what fits into the buffer, and nowadays, this is pretty much the only reason to use FIFO mode. A typical use case for example is a protocol, where almost all of the frames fits into 128B buffer, but in rare cases, the protocol must handle 512B long frames. In this article, the examples will present how to do that.

Another case when FIFO mode is required are some Direct and IQ modes. Setting up those are detailed in a [dedicated article](#), but FIFO handling is the same as documented here.

Advanced Packet Mode Features

If you have to look into the packet during reception (e.g., because you will use [RAIL_SetFixedLength\(\)](#) to handle a frame length coding which is not supported by hardware), you shouldn't use FIFO mode. Instead, you can use [RAIL_PeekRxPacket\(\)](#) to look into the packet, but still keeping it in the buffer, and only downloading it when you got the full frame.

If you need longer packets, you can customize both the Tx and Rx FIFO length, via [RAIL_SetTxFifo\(\)](#) and from the callback [RAILCb_SetupRxFifo\(\)](#).

If you want to have multiple frames in the buffer, there's also no reason to use FIFO mode: you can handle multiple frames without problems using packet mode. Before transmit, you can use the same APIs ([RAIL_SetTxFifo\(\)](#) and [RAIL_WriteTxFifo\(\)](#)) to write multiple packets, while you can store the packetHandles in Rx mode and keep them in buffer using [RAIL_HoldRxPacket\(\)](#) (except with DMP, but that would make the use of FIFO mode difficult as well).

Differences in FIFO Mode

Switching to FIFO mode doesn't actually do anything on the transmit side (since RAIL 2.0), but changing the receiver to FIFO mode slightly changes the behavior of the receive buffer: [RAIL_GetRxPacketInfo\(\)](#) (the usual way to access the received packet) won't work. You must use [RAIL_ReadRxFifo\(\)](#). Also, RAIL won't provide information on the frame length, and you should be careful to read exactly as many bytes as your frame; otherwise you might corrupt the next packet: E.g., if you leave the end of the packet in the FIFO, you will probably download it as the beginning of the next packet, corrupting that frame as well.

Enabling FIFO Mode

The easiest way to enable FIFO mode is on the **RAIL Utility, Initialization** component:

RAIL Utility, Initialization (inst0)

[Documentation](#)

Data Format Configuration

TX Data

Source of TX Data

Use frame hardware to packetize data ▼

Method of Providing TX Data

FIFO Mode ▼

RX Data

Source of RX Data

Use frame hardware to packetize data ▼

Method of Retrieving RX Data

FIFO Mode ▼

This calls `RAIL_ConfigData()` during init with the following parameters:

```
static const RAIL_DataConfig_t rail_data_config = {
    .txSource = TX_PACKET_DATA,
    .rxSource = RX_PACKET_DATA,
    .txMethod = FIFO_MODE, //doesn't do anything
    .rxMethod = FIFO_MODE,
};
RAIL_ConfigData(rail_handle, &rail_data_config);
```

However, to properly use FIFO mode, you probably want to know when the FIFO fills up/runs out of data. This is configured as below:

```
RAIL_SetTxFifoThreshold(rail_handle, TX_BUFFER_SIZE / 10);
RAIL_SetRxFifoThreshold(rail_handle, RX_BUFFER_SIZE - RX_BUFFER_SIZE / 10);
RAIL_ConfigEvents(rail_handle, RAIL_EVENT_TX_FIFO_ALMOST_EMPTY|RAIL_EVENT_RX_FIFO_ALMOST_FULL,
    RAIL_EVENT_TX_FIFO_ALMOST_EMPTY|RAIL_EVENT_RX_FIFO_ALMOST_FULL);
```

This sets the thresholds to 10% from being empty/full (for Tx and Rx, respectively), which is usually a good value, although you might want to increase it at high data rates (RX_BUFFER_SIZE should be 512B, if you did not implement `RAILCb_SetupRxFifo()`). Finally, it enables the two "threshold reached" events.

Using FIFO Mode: Transmit

On the transmit side, the idea is to fill the buffer completely, start the transmission, and fill the remaining when needed:

```
#define PACKET_LENGTH 1024
uint8_t packet_to_send[PACKET_LENGTH];
volatile uint16_t tx_offset;
void start_tx(){
    tx_offset = RAIL_WriteTxFifo(rail_handle, packet_to_send, PACKET_LENGTH, false);
    RAIL_StartTx(rail_handle, 0, RAIL_TX_OPTIONS_DEFAULT, NULL);
}

void sl_rail_util_on_event(RAIL_Handle_t rail_handle, RAIL_Events_t events){
    if ( (events & RAIL_EVENT_TX_FIFO_ALMOST_EMPTY) && (tx_offset < PACKET_LENGTH) ){
        tx_offset += RAIL_WriteTxFifo(rail_handle, packet_to_send+tx_offset, PACKET_LENGTH-tx_offset, false);
    }
}
```

This is just the related code; you probably have more events handled. If you look at the calls of [RAIL_WriteTxFifo\(\)](#), you can see that we always try to write the whole (remaining) packet, and do not check how much space we have in the FIFO. We could do that, using [RAIL_GetTxFifoSpaceAvailable\(\)](#), but it's not necessary. [RAIL_WriteTxFifo\(\)](#) will automatically stop when the FIFO is full, and return with the number of bytes successfully written. We do the same in the event, but we're adjusting both the source and the size with `tx_offset`. The condition `tx_offset < PACKET_LENGTH` is important, because the FIFO events do not check packet lengths. Unless you have multiple packets in the buffer, you'll get an [RAIL_EVENT_TX_FIFO_ALMOST_EMPTY](#) every time when you have just threshold amount of bytes remaining of your packet (since it's normal to have an empty buffer if you don't want to send anything).

Using FIFO Mode: Receive

The receive side is basically the opposite of the transmit side:

```
#define PACKET_LENGTH 1024
uint8_t packet_to_receive[PACKET_LENGTH];
volatile uint16_t rx_offset = 0;

void RAILCb_Generic(RAIL_Handle_t rail_handle, RAIL_Events_t events){
    if ( events & RAIL_EVENT_RX_FIFO_ALMOST_FULL ){
        rx_offset += RAIL_ReadRxFifo(rail_handle, packet_to_receive+rx_offset, PACKET_LENGTH - rx_offset);
    }
    if ( events & RAIL_EVENTS_RX_COMPLETION ){
        if ( events & RAIL_EVENT_RX_PACKET_RECEIVED ){
            if ( rx_offset < PACKET_LENGTH ){
                rx_offset += RAIL_ReadRxFifo(rail_handle, packet_to_receive+rx_offset, PACKET_LENGTH - rx_offset);
            }
            RAIL_RxPacketDetails_t packet_details = {
                .isAck = false,
                .timeReceived = {
                    .timePosition = RAIL_PACKET_TIME_AT_SYNC_END,
                    .totalPacketBytes = PACKET_LENGTH,
                },
            };
            RAIL_GetRxPacketDetails(rail_handle, RAIL_RX_PACKET_HANDLE_NEWEST, &packet_details);
        }
        rx_offset = 0;
    }
}
```

We start with an empty `packet_to_receive`, and the first thing we receive will be a single or multiple [RAIL_EVENT_RX_FIFO_ALMOST_FULL](#) (alternatively, [RAIL_EVENT_RX_SYNC1_DETECT](#)/[RAIL_EVENT_RX_SYNC2_DETECT](#) could be handled to set up `packet_to_receive` for reception).

Each time we get that event, we download as much from the FIFO as we can, and adjust the offset (again, we could check the amount of data in the FIFO using [RAIL_GetRxFifoBytesAvailable\(\)](#), but it's not necessary). When we get the [RAIL_EVENT_RX_PACKET_RECEIVED](#), we probably still have some data remaining in the FIFO, so we download that.

Then we get the `packet_details`, just like we do for a normal packet. **This step is very important** if you can't reset the FIFO between packets. If you leave the appended info in the FIFO, it will remain there and will be difficult to figure out where the next packet starts.

Finally, we reset the offset to be ready for the next packet, both on successful receive and errors.

Since this was a fixed length setup, there was no need to figure out the length. In variable length mode, you probably want to download the length field only first, set up a global variable with the length, and continue just like the above sample.

API Introduced in this Article

Functions

- [RAIL_PeekRxPacket\(\)](#)
- [RAIL_ReadRxFifo\(\)](#)
- [RAIL_ConfigData\(\)](#)
- [RAIL_SetTxFifoThreshold\(\)](#)
- [RAIL_SetRxFifoThreshold\(\)](#)
- [RAIL_GetTxFifoSpaceAvailable\(\)](#)
- [RAIL_GetRxFifoBytesAvailable\(\)](#)

Types and Enums

- [RAIL_DataConfig_t](#)
- [RAIL_DataMethod_t](#) (FIFO_MODE/PACKET_MODE)
- [RAIL_EVENT_TX_FIFO_ALMOST_EMPTY](#)
- [RAIL_EVENT_TX_FIFO_ALMOST_FULL](#)

Low Duty Cycle Theory

Low duty Cycle Mode

Introduction and Motivation

Low Duty Cycle (LDC, sometimes just Duty Cycle) by definition means switching something periodically on and off. However, in the context of low power radios, we usually use it for receive mode only. In this case, the radio is switching between a low power, long duration sleep state and short active states, wherein the radio is in receive state looking for valid preamble. In this way, the radio is able to save a huge amount of power.

Furthermore this operation is automatic, meaning no MCU intervention is needed for scheduling receive intervals, saving more energy. Typically an interrupt line or events are used to indicate a valid packet's arrival.

Synchronous and Asynchronous Duty Cycling

Duty cycling radio protocols can be sorted into asynchronous or synchronous protocols. Asynchronous means that the receiver doesn't know when the transmitter would send a packet, while in synchronous mode. As its name implies, there is temporal synchronization between the two connected parts; the messages will be sent at a specified time.

Some widely used protocols use LDC (e.g. Bluetooth low energy) in synchronized networks. By spending energy and time for synchronization, the receiver can receive all the packets while it is in sleep state most of the time.

However, in some cases, the overhead required by the synchronization doesn't worth it, e.g. if you need to receive messages from a transmitter that rarely sends anything. In such cases, you probably want to use asynchronous duty cycling on the receiver, but in such cases the transmitter cannot predict when the receiver is listening. The goal of this KBA is to solve this problem, and explain **how asynchronous LDC works**.

In this document, the theoretical configuration of LDC will be illustrated with application examples (general use cases), and at last some implementation notes will be shown for both EZRadioPRO and EFR32 devices.

Theory of LDC

The main idea behind the LDC is to move the power consumption of Rx to the Tx side. Though in LDC mode, both side are strongly dependent to each other focus on the Rx side first and assume an optimal Tx.

Rx Side Considerations - Preamble Detection Period

To minimize the receiver's side average power consumption, the time spent in active state should be minimized. What's the minimum time to keep the receiver on? When and how could the receiver detect an incoming message? What is this event? There are [more options](#) for the receiver to detect an incoming message, but we concentrate only to the preamble detection case.

In preamble detection duty cycle mode the radio periodically wakes up for a short duration to look for a preamble. When a valid preamble is detected, the radio remains in receive state until the packet is fully received (or an error detected during reception).

Note that the original goal of the preamble is to achieve bit synchronization of the incoming signal (or to restore the clock signal in other words). Therefore, the preamble should always be at least as long as it needs to be to reliably achieve this task. Most radio's datasheets ([Si4460/1/3/4](#) p. 28.) recommend selecting preamble length and preamble detection threshold (aka. preamble detection period), which is a minimal value to achieve 0% PER.

Define the following receiver time terms:

- **Tpdp**: preamble detection period

- **T_{ron}**: receiver's active state time (it should be longer, than the preamble detection period)
- **T_{sleep}**: receiver's sleep state time

and ignore other timing constraints, like wake-up time, radio state transition time, etc.

From this point, assume optimal Rx, and focus on the Tx side.

Tx Side Consideration - Application Examples - LDC in Asynchronous Networks

There is tradeoff between the Rx side's and the Tx side's power consumption: longer sleep period infer longer transmission time (see the examples section).

There are two widely used scenario to ensure package reception on the receiver side in asynchronous networks, one called as "Duty cycle" mode, and the other is the "Master-slave" or "Master blast" mode.

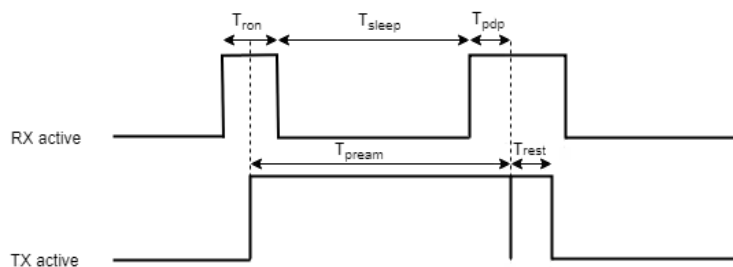
Duty-cycle Mode

This mode requires a packet with a very long preamble at the transmitter side. For the simplest example, assume the active time (T_{on}) and the sleep period (T_s) are given. In this case, the transmitter should have at least

$$T_{pream} = 2 * T_{ron} + T_{sleep}$$

long preamble to make sure all packets will be received, independent of the receiver state when the transmission started. The rest of the packet is transmitted in **T_{rest}** time.

In the worst case scenario, the transmitter starts transmitting slightly after the receiver wakes up, and does not provide long enough preamble for detection (T_{pdp}) in this wake-on period, continues the transmission during the receiver is in sleep state (T_{sleep}), and for the next active period ($T_{pdp} + T_{rest}$), where the receiver finally detects the packet. This is shown in the picture below.

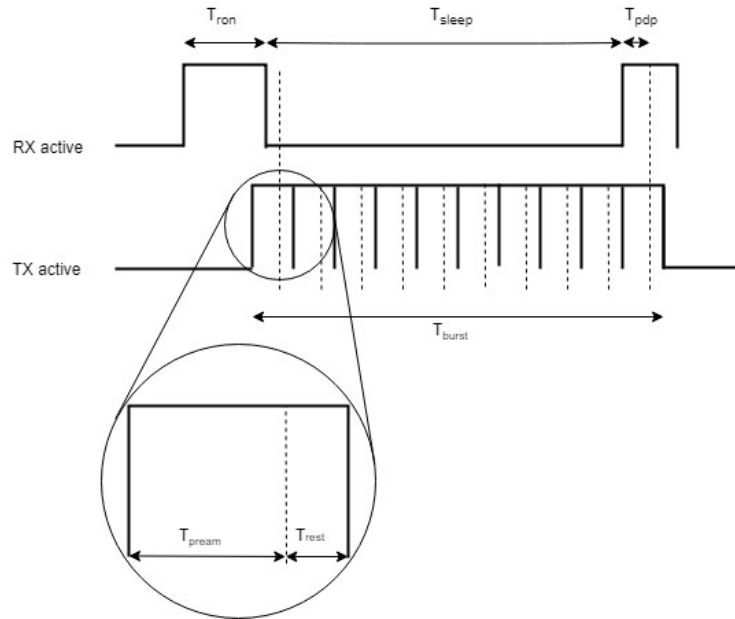


Compared to the case when the receiver is always in receive state, Duty-cycle mode saves approximately the duty-cycle proportion of power on the receiver side.

(Unsynchronized) Master-slave Mode

The master term refers to the transmitter and the slave term to the receiver(s).

This mode has a similar idea, but instead of one packet with long period it uses transmit burst with short packets back to back, which transmit time, **T_{burst}**, should fit to T_{pream} determined in the Duty-cycle mode (the preamble length is much shorter in this mode but it has to fit to T_{pdp} time at least, and we will call it T_{pream}). Usually the rest of the packet (syncword, header, payload, CRC, etc.) is for T_{rest} time in the air.



The slave must be in active state at least $2 * T_{\text{pream}} + T_{\text{rest}} = T_{\text{on}}$ long, which is higher than in the Duty-cycle mode for ensuring packet reception for every case.

Comparison of Applications

The downside of Duty cycle mode comes up when the radio wakes on the beginning of the preamble. This implies a long active period due to waiting for the whole packet. Furthermore, the radio has to handle if the preamble is lost, then reacquired due to interference. Finally, it's fairly easy for a malicious source to cause problems; it's easy to generate an infinite preamble.

On the other hand, with Master-slave mode the receiver only needs to be active for as long as it receives a single, short packet. The downside of Master-slave mode is that the receiver must be in active state as long as the longest possible packet is; otherwise it could miss the preamble of the packet.

Therefore the selection of application strongly depends on the protocol, or rather, the used protocol needed to define the LDC application mode.

Possible Improvements in Master-slave Mode

With Master-slave mode, it is possible to improve the performance and energy consumption. For example, the transmitter could check for an ACK after each transmit. If the transmission was acknowledged, the transmitter can stop. The drawback with this is that this only works with a single receiver, and the active period of the receiver should be extended to include the time the transmitter spends to wait for an ACK.

Another method to improve Master-slave mode is to use a specific "wake-up" message, and not the actual data. This wake-up message only tells the receiver when the actual data will be transmitted and the receiver can wake-up again for that. This worsens the power consumption on both receiver and transmitter side, but the data length doesn't affect the active time on the slave any more.

Note About Preamble Detection Selection

Low detection threshold may lead to misdetection and sensitivity loss, but halting the receiver longer in active state increases energy consumption.

Therefore the threshold should be good enough to achieve the required packet error rate. Note that the data rate may also impact on the threshold selection.

Implementation on EZRadioPRO

EZRadioPro devices have a low frequency (32 KHz) clock system, called as WUT (Wake-Up Timer). It can be operated from an external or an internal clock source (see in the API documentation for **GLOBAL_CLK_CFG** register, **CLK_32_SEL** bit fields). The radio uses this clock during sleep state for generating wake-up signal (the WUT activity distinguishes between sleep and standby states), thereby WUT supports LDC mode (and Low Battery Detection as well) by hardware.

Registers with **GLOBAL_WUT** prefix related to this timer. Both the sleep period and the active period (documentation calls it LDC period) should be set with a mantissa (**WUT_M** and **WUT_LDC** bitfields) and a common exponent (**WUT_R** bitfields). The active period may be less than the configured LDC period, when a packet arrived before its expiration (as shown in the second picture). We signed LDC period as Tron and sleep period as Tsleep.

Next to the periods, the preamble detection threshold should be configured. It determines the number of valid preamble bits the radio must receive to qualify a valid preamble. A shorter Preamble Detection Threshold may be chosen if occasional false detections can be tolerated. This threshold can be set through the **PREAMBLE_CONFIG_STD_1** register's **RX_THRESHOLD** bitfields, and all the other preamble related settings are located in the 0x10 Group of the registers.

After issuing a **START_RX** command, the radio's high frequency crystal should be started up, and the PLL requires some calibration time (see [AN585](#) p.1). This should be taken into consideration to set the receive time correctly.

Further documentation:

- LDC EZRadioPRO can be found in the [AN585](#), which describes Master-slave mode more in depth.
- [How does preamble detection threshold work.](#)
- [Detection of the absence of a signal.](#)
- [How to set Tx preamble length and preamble detection period](#), for a frequency hopping application, but it can still be used for an LDC application as well.

Implementation on EFR32

EFR32 devices have LDC (although this feature called as Duty Cycle mode) implemented as part of Channel Hopping feature. The theory is similar to that, which was described for EZRadioPRO devices, but here is a simpler implementation, only two relevant APIs should be known:

- [RAIL_ConfigRxDutyCycle\(\)](#): configures the mode and timing for the Duty Cycle operation
- [RAIL_EnableRxDutyCycle\(\)](#): enables radio's receiver ([RAIL_StartRX\(\)](#) or [RAIL_ScheduleRx\(\)](#) API should be called after this).
- [RAIL_EVENT_RX_DUTY_CYCLE_RX_END](#): event triggered after a complete active-sleep period, where ([RAIL_StartRX\(\)](#) or [RAIL_ScheduleRx\(\)](#) should be called again

For complete documentation, visit the [API documentation](#).

Note that the APIs do not put the MCU itself into sleep state. The easiest way to do that is to use the Power Manager service, which is fully supported by RAIL. This is documented in the [Power Manager article](#).

Notes:

- The [RAIL_ConfigRxDutyCycle\(\)](#) and the [RAIL_EnableRxDutyCycle\(\)](#) APIs are not supported on the EFR32XG1 family of chips, and in multiprotocol applications.
- The [RAIL_ConfigRxDutyCycle\(\)](#) and the [RAIL_EnableRxDutyCycle\(\)](#) APIs must not be called while the radio is on.
- There is a Duty Cycle example in the RAIL SDK, demonstrating the APIs above.

Thread Safety

Interrupt and Thread Safety in RAIL

In embedded software development, some of the most complicated debug challenges are caused by calling non-reentrant functions in interrupt context. Hence, it's in the developer's best interest to carefully design their application to avoid these scenarios.

To do so, however, requires sufficiently detailed knowledge of the interrupts and API functions - which are not accessible in a closed-source product like RAIL. This document aims to provide the information required to develop interrupt- and thread-safe applications in RAIL.

Thread Safety

In an application without a task scheduler, only an interrupt request can interrupt the main program. If you use a preemptive scheduler, like the scheduler available in most embedded OSes, higher priority tasks can interrupt lower priority tasks as well. Regardless, when looking at the RAIL APIs, the same concerns are present in either case:

- Is it safe to interrupt this API?
- Is it safe to call this API from a thread/interrupt which interrupted something?

The Event Handler

RAIL uses an event handler, which is set up by `RAIL_Init()`. In our examples, it's usually called `sl_rail_util_on_event()`. This function is called by the RAIL library, and it's **almost always called from an interrupt handler**. This means the event handler should be used with care:

- It should be kept in mind that interrupts are disabled when the event handler is running, so the function must not take long to return.
- More importantly, the function might be interrupting the main loop (or some other task).

Note that the first point above might not be completely true if interrupt priorities are used, in which case only interrupts at the same and lower priorities are disabled. However, the event handler will never be interrupted by another event handler as all RAIL interrupts must be used at the same priority.

General Rules for the RAIL API

First, let's collect the general rules of the API, and we'll detail exceptions in later points:

1. Calling any RAIL API from the main thread (or a single OS thread) is **safe**.
2. Calling any API from multiple threads is **unsafe**, except for DMP.
3. Calling most APIs from an interrupt handler is **safe** (see exceptions below).

Dynamic Multiprotocol (DMP)

In general, if you have a multi-threaded application, you should use RAIL from a single thread. The exception to this guidance is DMP, where in most cases each protocol runs in its own thread. In this scenario, using RAIL from each thread is safe, as each protocol has its own `rail_handle`. So, a more generalized wording of rule 2 is:

Calling any API from multiple threads is only safe if each thread has a dedicated `rail_handle`, and each thread only accesses RAIL with its own handle.

The few APIs that don't use `rail_handle` - like `RAIL_GetTime()`, `RAIL_Sleep()`, or `RAIL_Wake()` - can be called from any thread.

Interrupt Safety in General

In general, calling an API which changes the radio state (i.e., between Rx, Idle and Tx) can be risky. The simplest way to write interrupt safe application is to not call state changing APIs from any interrupt handler, including the RAIL event handler. This can be achieved by setting a flag or changing a state variable in the event handler instead of calling an API directly:

```
typedef enum {
    S_IDLE,
    S_START_RX,
    S_START_TX,
} state_t;

volatile state_t state;
volatile RAIL_Time_t last_event;

int main(){
    //init code

    state = S_START_TX;
    while(1){
        switch(state){
            case S_START_TX:
                RAIL_StartTx(rail_handle, 0, RAIL_TX_OPTIONS_DEFAULT, NULL);
                state = S_IDLE;
                break;
            case S_START_RX:
                RAIL_StartRx(rail_handle, 0, NULL);
                state = S_IDLE;
                break;
            default:
                break;
        }
    }
    return 0;
}

void sl_rail_util_on_event(RAIL_Handle_t rail_handle, RAIL_Events_t events)
{
    last_event = RAIL_GetTime();
    if ( events & RAIL_EVENTS_TX_COMPLETION ){
        state = S_START_RX;
        RAIL_SetTxPower(rail_handle, 200);
    }
}
```

Note that some RAIL APIs were called from the event handler, but none of those were state changing APIs.

Interrupt Safety with State Changing APIs

In some (usually time critical) cases however, it's not possible to avoid calling state changing APIs from the event handler (or other interrupt handler). State changing APIs are not always risky: Some APIs might be safe, as long as they don't interrupt another specific API.

Hence, in the following list, we identify the risky API after first specifying which initially-running (i.e., "interrupted") API makes it risky (and how). We've included in this list some interrupt combinations that might be "safe", but the end result is not predictable - i.e., the radio might be in Rx or in Idle, depending on which API is called first.

- Interrupting `RAIL_Start<something>()` with another `RAIL_Start<something>()` is risky, especially if they would start on different channels.
- Interrupting `RAIL_Idle(handle, <something>, true)` with any `RAIL_Start<something>()` is risky.
- Interrupting `RAIL_Idle(handle, <something>, false)` with any `RAIL_Start<something>()` is safe, but the end result is not predictable (i.e., the radio will either be in Idle, or start the requested operation).

- Interrupting `RAIL_Start<something>()` with `RAIL_Idle()` is safe but the end result is not predictable, and might cause strange events (see the next section for details).
- Interrupting `RAIL_StopTxStream()` with any `RAIL_Start<something>()` is **very** risky (the radio might remain in test configuration and start transmitting/receiving).
- Interrupting `RAIL_StopTx()` is safe. Interrupting `RAIL_StopTx()` with `RAIL_Start<something>()` is safe but the end result is not predictable (i.e., the radio will either be in Idle, or start the requested operation).
- Interrupting anything with `RAIL_StopTx()` is safe (see next section for important clarification). Interrupting `RAIL_StartTx()` with `RAIL_StopTx(handle, RAIL_STOP_MODE_ACTIVE)` is safe, but not predictable.
- Interrupting anything with `RAIL_StopTxStream()` is safe. Interrupting `RAIL_StartTxStream()` with `RAIL_StopTxStream()` is safe but not predictable.

RAIL_Idle in the Event Handler

Calling `RAIL_Idle()` or `RAIL_StopTx(rail_handle, RAIL_STOP_MODE_ACTIVE)` from the event handler might cause strange results. For example, let's say you're receiving on a channel and want to detect preambles using the event `RAIL_EVENT_RX_PREAMBLE_DETECT` and `RAIL_EVENT_RX_PREAMBLE_LOST`. The following scenario may unfold:

- Preamble lost interrupt is received, so (at least) other radio interrupts are temporarily disabled.
- You enter the event handler with `RAIL_EVENT_RX_PREAMBLE_LOST`.
- At this point, the radio detects a preamble. The interrupt is logged, but the handler cannot run since the interrupts are masked.
- Still in the event handler, you decide to turn off the radio with `RAIL_Idle(railHandle, RAIL_IDLE_ABORT, true)`.
- The radio turning off will generate a preamble lost interrupt.
- The radio is now off, and you return from the event handler.
- Interrupts are enabled again, so the pending preamble detect interrupt handler starts running.
- You enter the event handler with `RAIL_EVENT_RX_PREAMBLE_DETECT` and `RAIL_EVENT_RX_PREAMBLE_LOST` both set at the same time.

So you end up with a preamble detect event, even though the radio is **off**. This is usually harmless, since you always have the `_LOST` or `_ABORTED` event as well, but this demonstrates why your design must carefully consider in what order to handle events.

The easiest way to avoid this conflicted outcome is to disable the events that might cause problems when turning off the radio.

Another way to avoid this issue is to use `RAIL_Idle(rail_handle, RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS, true)`, which will clear the pending interrupts. However, using `RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS` has other drawbacks. It does force the radio state machine to idle state, and it might corrupt the transmit or receive FIFOs - in which case it must clear them, losing all data that might already be in there. It could also take more time to finish running than `RAIL_IDLE_ABORT`.

Critical Blocks

One usual way to avoid internal safety issues is to create critical (a.k.a. atomic, although in case of EFR32, their meaning can be different) blocks, in which interrupts are disabled, in the main thread to make sure some code segment is never interrupted. However, this can create other problems, so it should be used carefully. There's no general rule to avoid this kind of "collateral damage", but here's an example that should be avoided:

RSSI averaging is running, and just before it finishes, we interrupt it with `RAIL_StartTx()` which is called from a critical block. The following race condition could happen:

- We enter the critical block, interrupts are disabled.
- RSSI averaging done interrupt is received, but the interrupt handler won't start since interrupts are masked.
- StartTx turns off the radio, prepares it for transmit, then starts transmitting.
- We leave the critical block, interrupts are enabled again.
- RSSI averaging done interrupt handler runs at this point which will turn off the radio, aborting the current transmit.

One way to avoid the problem above is to clear interrupts in the critical block. This can be done by using `RAIL_Idle(handle, RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS, true)` at the beginning of the critical block, but the drawbacks of doing so

(mentioned above) should be kept in mind. In general, it's better to avoid risky interrupts without using critical blocks in the main thread.

Using FORCE_SHUTDOWN

In the two sections above we mentioned two usecases where `RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS` can be useful. In general however, `RAIL_IDLE` or `RAIL_IDLE_ABORT` is a sufficient and preferred way to stop transmitting/receiving - therefore the `FORCE_SHUTDOWN` modes should be only used when they are really needed (as in the specific scenarios described here). For more details, see the [article on the idle modes](#).

RAIL Config Files

Understanding RAIL Config Files

While this is a RAIL software documentation, it could be useful for a hardware engineer who's working on a radio setup as well. It helps if you have some knowledge of RAIL, but only basic C knowledge is a requirement.

Silicon Labs provides tools to generate RAIL config files, but it could be useful to understand how they work.

The Generated Files

The radio configurator is an advanced configurator (meaning it's not just an editor GUI for a header file) in Simplicity Studio v5, which generates 3 files into the `<project_dir>/autogen` folder:

- `rail_config.c`
- `rail_config.h`
- `radioconf_generation_log.json`

The Log

The file `radioconf_generation_log.json` provides some information on the generated output. For example, the bandwidth is usually not an input from the radioconf file, but a calculated value. That is reported in this file as `Acquisition Channel Bandwidth`.

The Header

The configurator generates `rail_config.h` containing a few lines apart from header comments and guards and includes:

```
#define PROTOCOL_ACCELERATION_BUFFER protocolAccelerationBuffer
extern uint32_t protocolAccelerationBuffer[];

#define RAIL0_CHANNEL_GROUP_1_PHY_STUDIO_868M_GMSK_500KBPS
#define RAIL0_CHANNEL_GROUP_1_PROFILE_BASE
#define RADIO_CONFIG_XTAL_FREQUENCY 39000000UL

extern const RAIL_ChannelConfig_t *channelConfigs[];
```

The first two lines define a layer to give access for the application to the `protocolAccelerationBuffer` variable. The `protocolAccelerationBuffer` is used for DMP switches, to accelerate channel hopping mechanism. This buffer can be removed in most cases, but the compiler should optimize it out if it is not used.

The `RAIL0_` macros provide some information to the application code on what is set up on the radioconf file.

`RADIO_CONFIG_XTAL_FREQUENCY` tells that the config was created for a 39 MHz crystal, while the last line externs a variable to access the configuration from the application.

The most important file, `rail_config.c`, will be analyzed in the subsequent sections. It's recommended to open a `rail_config.c` file for reference when reading this article.

Callback Functions

The configurator generates the following callback functions, with empty body:

```
uint32_t RAILCb_CalcSymbolRate(RAIL_Handle_t railHandle)
uint32_t RAILCb_CalcBitRate(RAIL_Handle_t railHandle)
void RAILCb_ConfigFrameTypeLength(RAIL_Handle_t railHandle, const RAIL_FrameType_t *frameType)
```

These callbacks are implemented in RAIL as weak functions, but were used only in a config generated by an older configurator. Since they're not needed with the new configurator, these functions override the internal RAIL functions saving code space.

The Config Array(s)

The largest part of the code is encoded into a config array (or arrays in case of a multi-PHY config), that looks like this:

```
const uint32_t ProtocolConfiguration_modemConfigBase[] = {
    0x01040FF0UL, (uint32_t) phyInfo,
    /* 0FF4 */ 0x00000000UL,
    /* 0FF8 */ 0x0003C000UL,
    // more address-value pairs
    0xFFFFFFFFUL,
};
```

These are essentially address-value pairs decoded and loaded to the correct place by RAIL.

Keep in mind that these are EFR32 *generation* specific arrays. For example, you cannot use an array on an xG13 device if it was generated for xG1. However, you can use the same config for devices from different families (MG, BG or FG) as long as the generation is the same. For more information, see [EFR32 Migration Guide for Proprietary Applications](#).

Helper Arrays

Some features, like the "3 out of 6" encoder, or the OOK dynamic slicer, need helper arrays that are referenced by a pointer in the main config array.

Some parts also need such helper arrays, e.g., xG23 generates a `hfxoRetimingConfigEntries` to configure minimal interference between the HFXO and the Local Oscillator of the receiver.

These arrays are usually passed either in the `phyInfo` or the main config array.

PHY Info Array

One of the values of the config array is a pointer to another array called `phyInfo`. This holds information of the config for RAIL. For example, the function `RAIL_GetBitrate()` returns the bitrate for the config. While it can be calculated from the config values, it's faster and even takes less code space to store it in the `phyInfo`.

IR Calibration Helpers

For details on the IR calibration, refer to the [Calibration article](#).

The radio configurator generates an array and a struct that is connected to the IR calibration, and they look like this:

```
static const uint8_t irCalConfig[] = {
    20, 41, 2, 0, 0, 49, 17, 0, 0, 0, 1, 0, 2, 100, 0, 1, 1, 47, 0, 0, 7
};

static RAIL_ChannelConfigEntryAttr_t channelConfigEntryAttr = {
    //a lot of #ifdef macros
};
```


The first one, `irCalConfig` tells RAIL how to run the IR calibration on this PHY. This array is passed as part of the **PHY Info Array**. The second one, `channelConfigEntryAttr` is used to cache the calibration result. Although it looks complex, it's really just a couple of `uint32_t`, depending on how many IQ receive and transmit paths the part have.

When you run `RAIL_CalibrateIr()`, `RAIL_ApplyIrCalibration()` or `RAIL_Calibrate()`, it will store the IR calibration result `channelConfigEntryAttr`. If you load a channel with the `channelConfigEntryAttr` value of `0xFFFFFFFFUL`, it will trigger the event `RAIL_EVENT_CAL_NEEDED`. This variable is passed as part of `RAIL_ChannelConfigEntry_t`.

Channel Entry Configuration

`RAIL_ConfigChannels()` takes a `RAIL_ChannelConfig_t` argument. It is defined in the config file, and usually looks like this:

```
const RAIL_ChannelConfig_t ProtocolConfiguration_channelConfig = {
    .phyConfigBase = ProtocolConfiguration_modemConfigBase,
    .phyConfigDeltaSubtract = NULL,
    .configs = ProtocolConfiguration_channels,
    .length = 1U,
    .signature = 0UL,
    .xtalFrequencyHz = 39000000UL,
};
```

The first argument is the base configuration: It is loaded immediately when you call `RAIL_ConfigChannels()`. The second parameter is part of RAIL 2.x multi-PHY features, we'll return to that later. The third parameter is a pointer to an array of `RAIL_ChannelConfigEntry_t`. The fourth parameter is the number of elements in the previous array. Signature is used only on modules, and the last parameter is the crystal frequency in Hz, obviously.

The channel entry is an array of structs, looking like this (with a single element):

```
const RAIL_ChannelConfigEntry_t channels[] = {
{
    .phyConfigDeltaAdd = NULL,
    .baseFrequency = 868000000,
    .channelSpacing = 1000000,
    .physicalChannelOffset = 0,
    .channelNumberStart = 0,
    .channelNumberEnd = 20,
    .maxPower = RAIL_TX_POWER_MAX,
    .attr = &entryAttr
#ifdef RADIO_CONFIG_ENABLE_CONC_PHY
    .entryType = 0,
#endif
#ifdef RADIO_CONFIG_ENABLE_STACK_INFO
    .stackInfo = NULL,
#endif
    .alternatePhy = NULL,
},
};
```

The first parameter is again part of the multi-PHY features. `maxPower` serves as a limit of the transmit power on some channels to meet regulatory standards. Note that it's in deci-dBm, so unless it's set to `RAIL_TX_POWER_MAX`, you'll need the **RAIL Utility, PA** component or the PA conversion functions implemented to convert it. `attr` points to the `entryAttr` struct, which is part of the IR calibration system, see above. The final 3 arguments are used by some of our stacks, and rarely used in proprietary setup.

The remaining parameters set the frequency and the channel numbers. The frequency of a given `channelNumber` is calculated with this formula:

```
channelFrequency = baseFrequency + channelSpacing * ( channelNumber - physicalChannelOffset )
```

Note that you cannot set the frequency beyond a limit without changes in the configuration. These limits depend on the part, so please rely on the configurator's help. If you set the frequency beyond the configuration's capability, you'll get a

`RAIL_ASSERT_FAILED_SYNTH_VCO_FREQUENCY` assert.

`channelNumberStart` and `channelNumberEnd` sets the limits of the entry, i.e., the first and the last channel.

API introduced in this Article

Types and Enums

- [RAIL_ChannelConfig_t](#)
- [RAIL_ChannelConfigEntry_t](#)

Introduction

Introduction to Multi-PHY and Multiprotocol

To get the most out of this article, some familiarity with the radio configurator is required, and it's also recommended that you understand the basics of radio protocol stacks (i.e. the [OSI model](#)).

With a customizable radio, the Wireless Gecko is a great option to support any radio configuration you need. You can set it up for standard IEEE 802.15.4, Bluetooth Low Energy, WMBus, or even the protocol used by your garage door opener. However, what if you need more than one of those standards? With single protocol radios, this requires a distinct 15.4 radio chip, a Bluetooth chip, and so on for each supported standard. Alternatively, with a single Wireless Gecko, you can store multiple "setups" in your device and switch between them. The only limitation is that it's still a single radio - you can't use multiple "setups" on the radio at the same time.

The best way to manage the sharing of this single radio resource depends on the particular use case targeted by your application. We provide different solutions (discussed below) to help you identify an optimal approach for your design.

Channel-based Multi-PHY

The most common use case in proprietary wireless is the need to support different PHY configurations. For example:

- You always handle received packets the same way, but you have to receive at multiple bitrates
- You have an asymmetric protocol, i.e. the Rx configuration is not the same as the Tx configuration
- You need wider bandwidth for CSMA/CA than your normal Rx mode to pass regulatory standards
- You have to handle different packet formats, but not at the same time: the upper layers of the stack decide which one to use
- Although the PHYs are technically the same, there is a power limitation on a given channel to pass regulatory standards
- The PHYs are technically the same, but the channel map is uneven - it's not possible to configure it using channel spacing

In these cases, Channel-based Multi-PHY is the best solution. It's called Channel-based, because in RAIL, you only need to change the channel to change the PHY. Channel-based Multi-PHY setups can be tested with RAILtest by selecting the channel with the `setChannel` serial command.

Protocol-based Multi-PHY

This is not as widely used as Channel-based Multi-PHY, but it's the next logical step. The typical use cases:

- DMP applications.
- Your device will support multiple regions, and you choose one during configuration, but the device will rarely switch between those.
- Your device will support multiple protocols, but the device will rarely switch between those.

For these, we recommend Protocol-based Multi-PHY. In this case, you can switch between protocols by calling `RAIL_ConfigChannels()`, which will load the whole configuration.

You can test your Protocol-based Multi-PHY setup using RAILtest. You can change between protocols using the `setConfigIndex` serial command.

Channel Change Performance

We generally recommend the Channel-based multi-PHY approach, especially when quick channel switch time is required.

If you have significant differences between channel groups, channel changing that involves a channel group change will take somewhat longer. In most cases, this is not too significant, but you should measure and calculate with it if you need short channel switch times (PRS channels can be used for measuring this time, see [the document about debugging](#)).

Also, if you have 3 channel groups, and only a single field is different between A and B, but C is very different compared to the other two, channel change between A and B will take the same time as channel change between C and A or C and B. This is because the radio configurator searches the common config between all 3 channel groups, and the remaining will be loaded on each channel group change.

If you have a lot of channel groups in the protocol, this could also cause a small but measurable delay during channel change. It takes some time for RAIL to look up the channel group which includes the requested channel. This is because RAIL does a simple search from the first channel group on every channel group change, for historical reason.

All of these delays are a result of the implementation in RAIL. See the [Understanding RAIL config files](#) article for more details.

You can find a setup guide for some of the mentioned Multi-PHY solution use cases in the [Multi-PHY Use cases and Examples](#) article.

Dynamic Multiprotocol (DMP) between RAIL-based Protocols

If you have full protocol stacks that are completely independent, it makes sense to develop the PHY and Link layer independently as well. These layers communicate with RAIL, and the cleanest solution to this problem is Dynamic Multiprotocol. Typical use case:

- One protocol is used to communicate with key fobs, while another is used to communicate in a home automation network

In Dynamic Multiprotocol, the RAIL scheduler decides which protocol is active.

Dynamic Multiprotocol is more capable than Multi-PHY, but also more complex, so it has some drawbacks when compared to Multi-PHY:

- Protocol change takes 450us on EFR32xG1x. This is a fixed worst-case delay, because the RAIL scheduler must plan time-sharing of the radio resource using calculations on a known sufficient time span in order to accommodate protocol switches. The time differs on Series 2 parts, but in general, longer than even protocol-based Multi-PHY, since it is set for worst case.
- It requires the multiprotocol version of the RAIL library, which has a bigger memory footprint (both RAM and code) than the single protocol version.

Dynamic Multiprotocol between RAIL-based and other Silicon Labs Stack Protocols

The use case for this is straightforward:

- If you need Bluetooth and a RAIL-based protocol in the same application

At the moment, Silicon Labs only provides such a DMP solution based on the Bluetooth stack and RAIL. You can access it by installing the Bluetooth SDK and starting one of the DMP applications. See [Dynamic Multiprotocol Development with Bluetooth and Proprietary Protocols on RAIL](#).

Other Multiprotocol Solutions

You can find other types of multiprotocol mentioned in some Silicon Labs documents, but they are rarely used:

- Programmable multiprotocol means multiple firmware images for each protocol, and the protocol is decided during the manufacturing process, i.e. a device is either programmed to be a Bluetooth or a Zigbee device.
- Switched multiprotocol means two firmware images, one for each protocol. A bootloader loads the firmware needed for the protocol, and the two firmware can communicate with each other in a shared NVM storage. Protocol switching in switched multiprotocol is rather slow, and using DMP is almost always a better choice.

- Concurrent multiprotocol means that the PHY and maybe the lower layers of the stacks are the same, but the upper layers are different. E.g. You can run Zigbee and Thread on the same channel at the same time.

Related Documentation

- [Dynamic Multiprotocol Development with Bluetooth and Proprietary Protocols on RAIL](#)
- [AN1253: EFR32 Radio Configurator Guide](#)
- [Multiprotocol Fundamentals](#)

MultiPHY Use Cases

Multi-PHY Use cases and Examples

In this article, we go through a couple of Multi-PHY examples and their setups using the Radio Configurator. If you haven't already, please read [Introduction to Multi-PHY and Multiprotocol](#) first to understand the various Multi-PHY and Multiprotocol capabilities of the Wireless Geckos. Some basic understanding of radio configs and the Radio Configurator is also required. You can find a more detailed description of the new Radio Configurator in [AN1253](#).

The article includes short videos to demonstrate how to set up the given use case.

Channel-based Multi-PHY

The main idea behind Channel-based Multi-PHY is to create channel groups, each of which can have different PHYs. Changing between these channel groups is handled by RAIL: from the RAIL API, you can load the required PHY just by selecting a channel.

A PHY configuration exposes elements of the radio configurator for each of the channel groups in the selected protocol. Channel groups can be added with the  icon. When you add an element to the channel-based configuration, it will initially inherit the PHY from the root protocol. Inside each channel group, the root protocol values can be overridden by toggling the "Customized" button or by selecting a predefined PHY.

Different Bitrate

Let's say you need to work on the same 915MHz frequency, but on both 100kb/s and 500kb/s (with deviation 50kHz and 250kHz, respectively). Typically, this is used as an extension of a protocol that originally only supported the lower bitrate: The default bitrate is 100kb/s, but it's possible to switch to 500kb/s after a handshake.

[VIDEO](#)

In the above setup, channel 0 uses 100kb/s bitrate, while channel 1 uses 500 kb/s. Note that the channels are on the exact same frequency, but their configuration is slightly different - you might call these *virtual channels*. From RAIL, you can change between these virtual channels just as regular channels. If you call `RAIL_StartRx(rail_handle, 0, NULL)` it will search for 100kb/s packets, while `RAIL_StartRx(rail_handle, 1, NULL)` will search for 500kb/s packets.

Different Packet

Let's say your protocol defines a variable length data packet (1st byte is length) and a fixed, 2B long ACK packet.

[VIDEO](#)

In the above setup, channel 0 is for data packets, while channel 1 is for ACK packets (both Tx and Rx) - on the same frequency, so these are *virtual channels* just like in the previous example. A transmitting device would send a data packet on channel 0, then start listening on channel 1 for ACK, while a receiving device would receive on channel 0, and transmit an ACK on channel 1 after successful reception.

Note that the only difference between the PHYs is the length decoding algorithm itself. On the configurator GUI, we have two separate and complete configurations, but when generated, the configurator will detect what the difference is, and only generate a few lines of difference.

Asymmetric Protocol (Wireless M-Bus T Mode)

Wireless M-Bus T mode defines a different protocol based on the direction:

- **Meter -> Other (M2O):** 868.95MHz, 3of6 encoding, 100kb/s

- Other -> Meter (O2M): 868.3MHz, Manchester encoding, 32.768kb/s

VIDEO

The above example implements the "Other device" (typically collector). It receives M2O on channel 0 and transmits O2M on channel 1. The implementation required two pre-defined PHYs without any further modifications.

Transmitting modeT M2O messages requires other tricks on the EFR32xG1x parts, see [Using RAIL for Wireless M-Bus Applications with EFR32](#) for details.

Wide Bandwidth for CSMA/CA

Some regulatory standards also standardize the bandwidth for CSMA/CA, but your protocol would have better sensitivity with a narrower bandwidth. The trick here is to understand that this is actually a subset of the Asymmetric protocol use case: Since CSMA/CA will listen on the same channel that you will use for transmitting and the bandwidth setting in the configurator only affects receive performance, you can define a virtual channel with your setup with the required wider bandwidth:

VIDEO

In the above example, channel 0 uses the bandwidth recommended by the configurator, while channel 1 has a fixed, wide bandwidth. You would receive on channel 0 with high sensitivity, and transmit on channel 1. The CSMA/CA preceding the transmit would also happen on channel 1 using the fixed, wide bandwidth.

You can check the bandwidths in the `autogen/radioconf_generation_log.json` file: 4.8kHz for channel 0 and 100kHz for channel 1.

Uneven Channel Spacing

Channel groups can be used in "single-PHY" mode as well: one example of that is setting up uneven channel spacing. Let's say you have to receive on 868.0MHz, 868.33MHz and 868.4MHz - so the channel spacing is 330kHz and 70kHz at the same time. You obviously can't configure this with a single channel spacing field, but you can with channel groups:

VIDEO

With the above setup, you will have channels 0, 1 and 2 on the required frequencies.

How Channel-based Multi-PHY Actually Works

See the Multi-PHY capabilities chapter in the [Understanding RAIL config files](#) article.

Protocol-based Multi-PHY

The main idea behind Protocol-based Multi-PHY is to create multiple - completely independent - configs, and use `RAIL_ConfigChannels()` to select the required one.

The order of protocols will be used in the generated `channelConfigs[]` array, i.e. the first protocol in the list will have index 0 and so on.

A protocol can be added to the config with the  icon.

Multiple Regions

Let's say your device operates on the 868MHz 500kb/s pre-configured PHY in Europe, and on the 915MHz 100kb/s pre-configured PHY in North America.

VIDEO

With the above setup, calling `RAIL_ConfigChannels(railHandle, channelConfigs[0], NULL)` will load the European config, while `RAIL_ConfigChannels(railHandle, channelConfigs[1], NULL)` will load the North American config.

Configuration for a DMP Setup

The above setup can also be used in a Dynamic Multiprotocol setup. You'll need two rail handles, let's call them `railHandle868` and `railHandle915`. During the initialization of the 868MHz protocol, you'll call `RAIL_ConfigChannels(railHandle868, channelConfigs[0], NULL)`, and during the initialization of the 915MHz protocol, you'll call `RAIL_ConfigChannels(railHandle915, channelConfigs[1], NULL)`. From this point, RAIL will automatically load the required config using the RAIL scheduler.

PA Configuration

RAIL will automatically change everything needed for Multi-PHY during channel or protocol change, except one thing: The PA (power amplifier) configuration. On Wireless Geckos, changing the PA is needed if you change between a sub-GHz and a 2.4GHz config (or maybe between a low power and a high power 2.4GHz config).

To solve this, you should pass a function pointer as the last parameter of `RAIL_ConfigChannels()`.

If you use the **RAIL Utility, PA** and **RAIL Utility, Initialization** component, you might want to set this to `sli_rail_util_on_channel_config_change()` - this will first call the function `sli_rail_util_pa_on_channel_config_change()` in the PA component, which selects the right PA, and then call `sli_rail_util_on_channel_config_change()`, what the application can implement.

If you don't use either of these components, you should create your own callback function, and call `RAIL_ConfigTxPower()` to select the right PA for the given config.

You can also use `RAIL_EnablePaAutoMode()` to enable automatic switching between the PAs or modes of the PA whenever you set the Tx power with `RAIL_SetTxPowerDbm()`. If you enable auto PA mode, you cannot use `RAIL_ConfigTxPower()`.

This is implemented in both **RAILtest** and **Simple TRX Multi-PHY**.

IR Calibration in Multi-PHY Configs

If you're not sure what IR calibration is, please refer to the [Calibration](#) article.

Configuration and Cache Variables in `rail_config.c`

This section is copied from the [Understanding RAIL Config Files](#) article.

The radio configurator generates two elements - an array and a struct - that are involved in IR calibration. They look like this:

```
static const uint8_t generated_irCalConfig[] = {
    24, 69, 3, 6, 4, 16, 1, 1, 1, 3, 1, 6, 0, 16, 39, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0
};

RAIL_ChannelConfigEntryAttr_t generated_entryAttr = {
    { 0xFFFFFFFFFUL }
};
```

The first one, `generated_irCalConfig`, tells RAIL how to run the IR calibration on this PHY. This array is passed as part of the PHY info array. The second one, `generated_entryAttr` is used to cache the calibration result. When you run `RAIL_CalibrateIr()`, `RAIL_ApplyIrCalibration()` or `RAIL_Calibrate()`, it will store the IR calibration result here. If you load a channel with the special entryAttr value of `RAIL_CAL_INVALID_VALUE`, it will trigger the event `RAIL_EVENT_CAL_NEEDED`. This is passed as part of the channel entry configuration.

IR Calibration in Protocol-based Multi-PHY

IR calibration should be performed for all protocols. This will be requested from the application through the `RAIL_EVENT_CAL_NEEDED` event. After the calibration is performed on each protocol, RAIL can use the cached results from these prior operations. When doing so, the calibration values will be applied automatically and near-instantaneously.

IR Calibration in Channel-based Multi-PHY

The radio configurator will automatically detect if channel groups are similar enough to use a common calibration (e.g. only the packet is different). In these cases, both `irCalConfig` and `entryAttr` will be the same. Calibration should be performed on each channel group that has its own `irCalConfig` and `entryAttr`. RAIL will detect this and request calibration through `RAIL_EVENT_CAL_NEEDED` if you select a channel group that has a different IR calibration setup and `RAIL_CAL_INVALID_VALUE` in `entryAttr`. Once calibration is performed on each channel group that needs it, RAIL can then cache these values for future use. As in Protocol-based Multi-PHY, with cached results the calibration values will be applied automatically and near-instantaneously.

Calibration on Multi-PHY at Startup

In some cases, using the `RAIL_EVENT_CAL_NEEDED` event to calibrate is not viable, and you might want to explicitly request calibration on everything at startup. To do that, you should:

- loop through all protocols
- loop through all channel groups that have their own `irCalConfig` and `entryAttr`

For the latter, you should check the generated `rail_config.c`. There's no easy way to figure out which channel to calibrate from code. To loop through the channel groups, you can use the API `RAIL_PrepChannel()` to trigger the PHY loading. The calibration code will likely look something like this:

```
#define NUMBER_OF_GROUPS 5
const int16_t calibration_channels[NUMBER_OF_GROUPS] = {0, 10, 25, 50, 65};

for(uint8_t i=0; i<NUMBER_OF_GROUPS;i++){
    RAIL_PrepChannel(rail_handle, calibration_channels[i]);
    RAIL_CalibrateIr(rail_handle, NULL);
}
```

In the above example, we have 5 channel groups that require calibration (`NUMBER_OF_GROUPS`). The first channel of each channel group is stored in `calibration_channels`.

If you have more protocols, you can simply load them using `RAIL_ConfigChannels()`. Note that the first channel of the protocol is automatically configured, so `RAIL_PrepChannel()` is only needed for Channel-Based Multi-PHY.

Multi-PHY in Connect

The Connect stack has some limitations on Multi-PHY features:

- It doesn't support Protocol-Based Multi-PHY: Connect always loads the first protocol (`channelConfigs[0]`)
- It doesn't support PA change, so Multi-PHY channels should use the same frequency band (i.e. either 2.4GHz or sub-GHz)

DMP

Dynamic Multiprotocol (DMP) in RAIL

Before reading this article, make sure to read the [Introduction to Multi-PHY and Multiprotocol](#), as DMP is often misunderstood and used in applications that could be better served by multi-PHY. We recommend that you (at least) browse through all the other training articles before reading this (DMP) document, since DMP affects most APIs in one way or another.

Note that this article will not provide you with all of the knowledge required for DMP development. Rather, it will summarize what's already documented, and highlight differences between RAIL single protocol and multiprotocol operations.

Some chapters below are not important if you need RAIL-Bluetooth DMP, but even in that case, most chapters are still relevant because they describe RAIL development in a DMP environment.

What is Dynamic Multiprotocol

Dynamic multiprotocol time-slices the radio and rapidly changes configurations to enable different wireless protocols to simultaneously operate reliably in the same application.

A typical use case involves a RAIL-based proprietary protocol and the Silicon Labs Bluetooth stack running on the same chip at the same time. Since Bluetooth communication is periodic and therefore predictable, other protocols can use the radio between Bluetooth communication bursts. However, it's also helpful to use DMP when all involved protocols are proprietary RAIL-based, to benefit from the code clarity provided by the separate RAIL instances.

From a firmware perspective, RAIL-DMP works on the principle that each protocol operates as is scheduled "alone" on the radio transceiver. A bit in a similar fashion that a process on the computer running and not being necessarily aware that other processes are scheduled at the same time.

Multiple RAIL protocol instances can be "created". Each instance can use the driver in the same way, and do so almost independently.

However, from a hardware perspective the chip has only a single instance of the radio. As such, RAIL-DMP is similar to multitasking: the same resource must be shared among multiple "tasks" demanding its time. This sharing is managed by the radio scheduler.

"Static" Multiprotocol

Static multiprotocol is a common requirement in IoT. For example, you only need Bluetooth for commissioning, and while Bluetooth is enabled, you don't need the main protocol running. For this use case, we also recommend DMP. However development is much simpler for it. You only need to initialize DMP and make sure to call `RAIL_Idle()` before switching protocols. So you only need to read the [Setting up a Project for RAIL-RAIL DMP](#) chapter here. Everything else is just recommended.

The Radio Scheduler

The radio scheduler's job is to give (or revoke) radio hardware access to (or from) the protocols running on it. To accomplish this, it implements cooperative, preemptive multitasking.

- It's cooperative: Protocols should tell the scheduler when and for how long they need the radio, and protocols should yield the radio when not using it anymore.
- It's preemptive: Protocols should define priority to each radio task, and the scheduler will abort tasks if a higher priority radio task must run.

Note that the radio scheduler schedules only radio tasks. Although DMP is often used with an RTOS (like FreeRTOS), **RTOS tasks and Radio tasks - despite the common name - are very different terms**. To make this less confusing, our documentation sometimes uses *radio operation* instead of *radio task*.

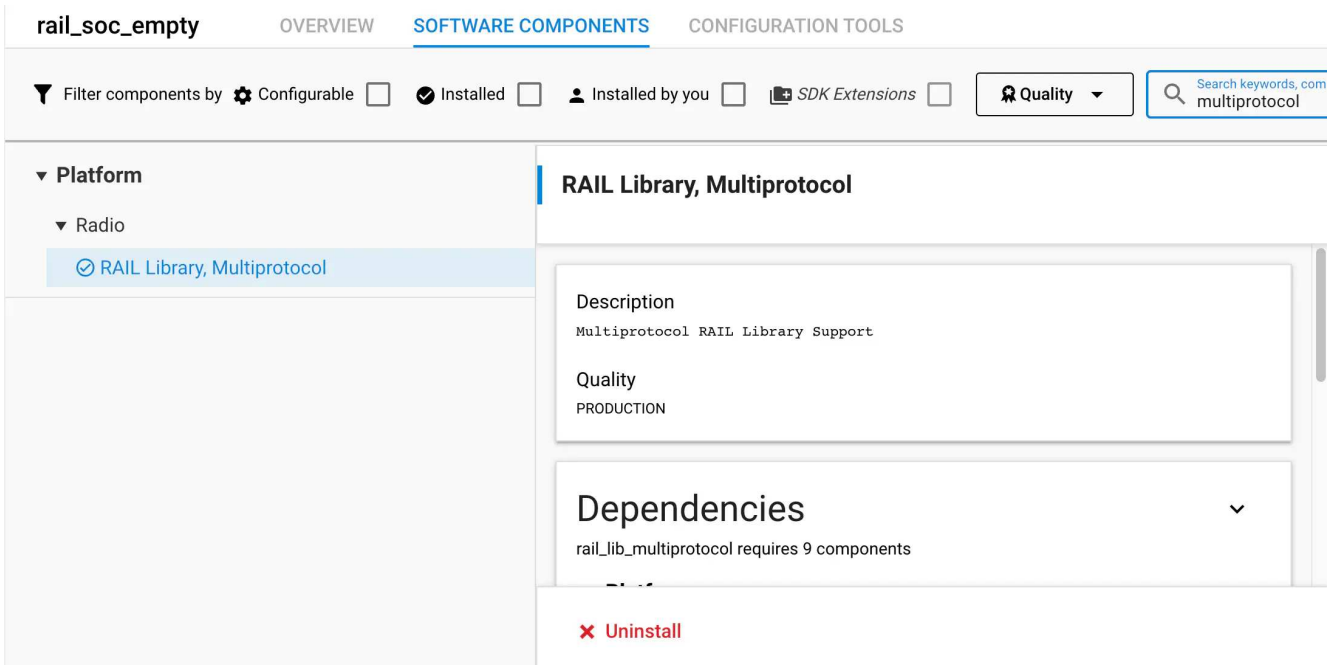
For the radio scheduler to calculate the time required for a given radio operation, it must know how much time it takes to change radio access from one protocol to another. Therefore, in DMP, **a protocol change always takes a fixed amount of time**. This is defined by `TRANSITION_TIME_US`, and could differ from part to part. As of writing this article, it is between 430-510us.

This transition time can be [tuned and overridden](#).

Setting up a Project for RAIL-RAIL DMP

If you need RAIL+Bluetooth DMP, follow [Dynamic Multiprotocol Development with BLE and Proprietary Protocols on RAIL](#). This chapter describes how to set up DMP in a RAIL-only project.

To use DMP, you should first switch to the multiprotocol RAIL library, under plugins. Since DMP has a bigger memory footprint (both RAM and code), it's provided in a library which is not enabled in single protocol examples.



The screenshot shows the 'rail_soc_empty' project in the 'SOFTWARE COMPONENTS' tab. The left sidebar shows a tree view with 'Platform' expanded, then 'Radio', and 'RAIL Library, Multiprotocol' selected. The main area displays details for the 'RAIL Library, Multiprotocol' component. It includes a 'Description' section with the text 'Multiprotocol RAIL Library Support', a 'Quality' section with the value 'PRODUCTION', and a 'Dependencies' section stating 'rail_lib_multiprotocol requires 9 components'. At the bottom, there is a red 'Uninstall' button.

Next, you'll need PHY configurations for your protocols. You can either use Protocol Based Multi-PHY (see [the Multi-PHY usecases and examples article](#) for details), or you can use embedded protocols, e.g. IEEE 802.15.4 2.4GHz 250kbps by calling `RAIL_IEEE802154_Config2p4GHzRadio()` or configuring the **RAIL Utility, Initialization** component accordingly.

RTOS

RTOS is not technically required for RAIL DMP, although it's highly recommended to clearly separate protocols. And for RAIL+Bluetooth DMP, we actually only support RTOS-enabled projects. Therefore, RTOS will not be discussed in this article; it doesn't impact how you use RAIL APIs in RAIL-only DMP.

RAM Allocation

RAIL requires RAM allocated for each initialized protocol, to store its state variables. Since GSDK v4.0, the multiprotocol RAIL library automatically allocates RAM for two protocols. For more than two protocols, call `RAIL_AddStateBuffer3(RAIL_EFR32_HANDLE)` before initializing the 3rd protocol (or earlier). For even more protocols, use `RAIL_AddStateBuffer4(RAIL_EFR32_HANDLE)` and the generic `RAIL_AddStateBuffer()`.

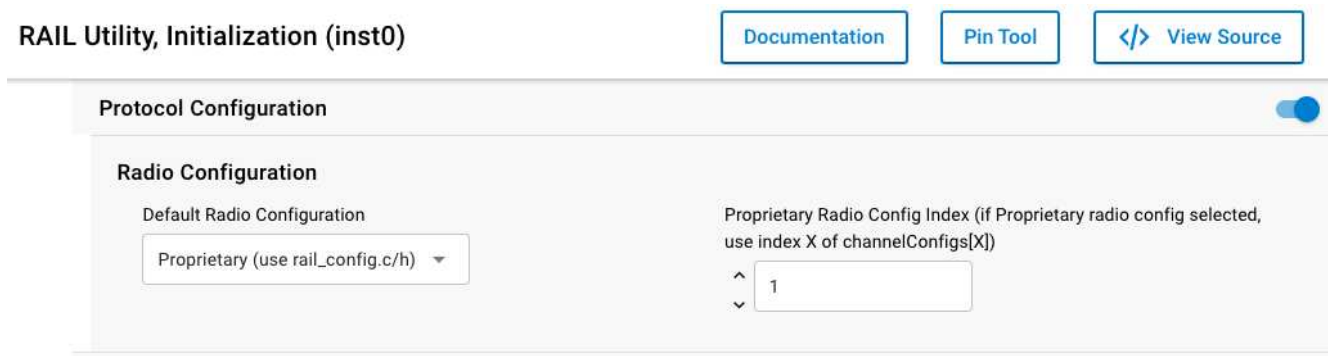
Note that `RAIL_EFR32_HANDLE` is a special RAIL handle, which can be used if a RAIL handle is needed before `RAIL_Init()`, like the above case.

Before GSDK v4.0, RAM was allocated in the application and needed to be passed to `RAIL_Init()`.

RAIL Initialization

Since GSDK v3.x, we recommend using the component **RAIL Utility, Initialization** to initialize RAIL. This is an instancable component; you can create multiple instances for RAIL-RAIL DMP.

It's a good idea to select the correct radio config for each instance using this component. E.g., depicted below, we selected the second protocol of the radio configurator, `channelConfigs[1]`. This is also the location where we could easily select IEEE 802.15.4 PHY.



If you wish not to use the component, initializing RAIL can be done exactly the same as single protocol. You will find some DMP related arguments, but those are ignored since GSDK v4.0 (they were used for memory allocation before 4.0).

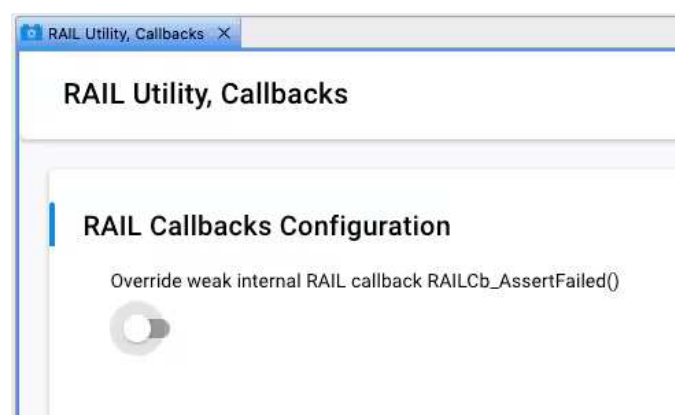
RAIL API Calls and Code Separation

RAIL APIs are the same for all protocols, and their `RAIL_Handle_t` argument decides which protocol is called. The RAIL handle can be obtained via `sl_rail_util_get_handle(SL_RAIL_UTIL_HANDLE_INST0)` where `INST0` is the name of the init component instance, with all uppercase.

It is highly recommended to implement each RAIL protocol in separate C files. Otherwise, it can be difficult to follow which call is intended for which protocol.

Assertion

RAIL assertion, i.e., the `RAILCb_AssertFailed()` should be common for all protocols. If you add RAIL next to BLE for example, you should disable this assert in your own protocol, as it is handled in the stack. This can be done in the **RAIL Utility, Callbacks** component.



Other APIs

Most of the RAIL APIs work exactly the same as in single protocol mode. However, you should be careful with:

- protocol independent APIs, i.e. the ones that don't have a `rail_handle` argument
- APIs that affect the state of the radio scheduler, i.e. schedule or stop a radio task

Protocol Independent APIs

There are a few APIs that always work the same way, regardless of which protocol they were called from:

- `RAIL_SetTime()` / `RAIL_GetTime()` : the timebase (which is also used for timestamping) is common for all protocols. It is not recommended to call `RAIL_SetTime()` in a DMP application (and calling it in a single protocol application can still be dangerous).
- `RAIL_ConfigMultiTimer()` : MultiTimer mode is a requirement in DMP, so it's enabled by default and cannot be disabled.
- `RAIL_Sleep()` / `RAIL_Wake()` : Sleep management should be handled protocol-independently (e.g. in the idle task of the RTOS).

There are some other APIs that don't require `rail_handle`, but they usually need a packet handler or timer handler, and so are still tied to a protocol.

Scheduling a Finite Radio Task

The following APIs will trigger the radio scheduler to schedule a finite length new radio task:

- `RAIL_ScheduleRx()`
- `RAIL_StartTx()`
- `RAIL_StartScheduledTx()`
- `RAIL_StartCcaCsmaTx()`
- `RAIL_StartCcaLbtTx()`
- `RAIL_StartAverageRssi()`

(Note that `RAIL_StartRx()` and `RAIL_StartTxStream()` are not listed, as these are **not finite** length tasks.)

All the above APIs have the optional argument `RAIL_SchedulerInfo_t`. In single protocol mode, this argument is ignored, and can be NULL. In multiprotocol mode, this is the base of the scheduling, with its 3 members:

- `priority` : 0 to 255, 0 being the highest. Equal priority does not preempt
- `slipTime` : maximum time the task can be delayed due to other protocols, in us
- `transactionTime` : the time the task takes, in us

When the radio hardware is available, none of these scheduler info arguments are used. However, if two tasks are scheduled at the same time, the radio scheduler will try to stagger the operations using `slipTime` in order to satisfy both tasks. If that fails, the lower `priority` task will not get access to the radio.

It is also possible that an application schedules a high priority radio task when a conflicting lower priority task has already started. In that case, the lower priority task will be aborted (possibly in the middle of frame transmission).

If `RAIL_SchedulerInfo_t` is NULL, the scheduler will assume all parameters to be 0, i.e. highest priority, with no `slipTime`. It is highly recommended to **not** use this feature, and it's only mentioned here to simplify debugging.

Each protocol can schedule at most a single finite task.

For more details on the radio scheduler, see the [Dynamic Multiprotocol User's Guide](#) and the [Multiprotocol description in the RAIL API doc](#).

Yielding

After each radio task, the application **must** yield the radio. That can be achieved by calling `RAIL_YieldRadio()` or `RAIL_Idle()`. The latter also turns off the radio, which should have already occurred for finite tasks. Manual yielding is required because

the scheduler doesn't know about any automatic follow-up tasks (like ACKs). You might want to yield immediately after receiving a packet, or you may instead want to wait for the auto ACK to go out.

Since yielding the radio will also call the radio scheduler to start/schedule the next radio task, you must first capture everything you need that might be overwritten by another protocol after yield, even in interrupt context. For example, timestamps of a transmitted packet must be accessed before yielding.

A Tx operation scheduled in the future can be cancelled using [RAIL_Idle\(\)](#) or [RAIL_StopTx\(\)](#), where the latter doesn't yield the radio. An Rx operation scheduled in the future can only be cancelled using [RAIL_Idle\(\)](#).

Background Rx

[RAIL_StartRx\(\)](#) receives special handling in the radio scheduler, as it schedules an infinite task - namely, background Rx. The main difference when compared to finite tasks is that an infinite task sets the "default" state of the radio. It can be aborted by higher priority finite tasks, but it will be resumed automatically after the higher priority task is finished. For background Rx, this practically means that the radio will automatically return to Rx after Tx or packet reception, and you don't have to use auto state transitions like you do in single protocol mode. (Currently, the only infinite task supported by RAIL is background Rx).

Keep in mind that RAIL does not store the channel of background Rx. If you interrupt background Rx with a finite task on a different channel, the channel of the background Rx will change as well. E.g. if you're receiving in channel 0, then transmit on channel 1, the radio will "return" to receiving on channel 1. To change the background Rx channel back to 0, simply call [RAIL_StartRx\(\)](#) again after you yield the radio.

[RAIL_StartRx\(\)](#) has the same [RAIL_SchedulerInfo_t](#) configuration as the finite tasks, but depends only on the `priority` member - which, in general, should be the lowest priority used by the protocol.

Background Rx can be aborted using [RAIL_Idle\(\)](#), which also yields the radio. In DMP mode, it's recommended to only use [RAIL_Idle\(\)](#) to stop background Rx.

It is also possible to change the priority of the background RX using [RAIL_SetTaskPriority\(\)](#) - this is useful e.g. to increase the priority during packet reception, or after a particular packet.

TxStream

[RAIL_StartTxStream\(\)](#) is called a debug task, which must have the highest priority, because it can't be aborted. Otherwise, it works similarly to infinite tasks.

This API is different from all others as it doesn't use [RAIL_SchedulerInfo_t](#). Since a stream can't really be aborted, the radio scheduler will handle this with the highest priority and no slip time.

Stream can be stopped with [RAIL_StopTxStream\(\)](#), [RAIL_Idle\(\)](#), or [RAIL_YieldRadio\(\)](#), although it is recommended to use [RAIL_StopTxStream\(\)](#) for clarity.

Auto State Transitions

You might wonder, "what's the point of auto state transitions in DMP, if background Rx changes the default state to Rx?" The added value of auto state transitions is that the resulting state will inherit the original state's priority.

For example, you probably want the ACK reception on the same priority as the transmission itself, so you can set [RAIL_SetTxTransition\(\)](#) to set up receive after the transmission for the ACK, and configure a timeout for it using [RAIL_SetStateTiming\(\)](#).

If you have automatic state transition set up, you should only yield the radio after the whole operation (e.g. ACK reception or timeout) is finished. It's highly recommended to have automatic timeouts to prevent a high priority infinite task, but if you haven't set up timeouts, you can cancel the receive and yield the radio using [RAIL_Idle\(\)](#).

To summarize:

- In RAIL single protocol, the default state is always idle, and auto state transitions should be used to always return the radio to Rx. In DMP, background Rx should be used for this.

- In single protocol mode, Rx state started by auto state transition is the same as Rx state started by StartRx. In DMP, auto state transitions are intended for ACK, as they inherit the (usually high) priority of the preceding task.

For more details, see the [Dynamic Multiprotocol User's Guide](#).

MultiTimers

RAIL includes a timer virtualization service, called MultiTimer. Since its memory footprint is not negligible, this feature is disabled by default in single protocol mode. However, in multiprotocol mode, it is always enabled, since multiple timers are required for the multiple protocols. Therefore, using MultiTimers in your protocol implementations has no drawback in DMP.

FIFOs

DMP maintains separate Tx FIFO memory locations for the used protocols; however, the Rx FIFO is common for all of them. This means that the Rx FIFO is cleared during protocol switch, to avoid information leaking from one protocol to the other. This also means that the FIFO **must** be downloaded before yielding the radio. In practical terms, this means that it's recommended to download received packet in the event handler, and the use of the [RAIL_HoldRxPacket\(\)](#) API should be avoided.

Error Handling

In single protocol RAIL, you generally need to implement the following error handling:

- `RAIL_StartXYZ()` / `RAIL_ScheduleXYZ()` might return an error. If no error were returned, the operation was either finished, or an event will be triggered.
- The event handler, where either a success or error event can be triggered, e.g. [RAIL_EVENT_TX_PACKET_SENT](#) or [RAIL_EVENT_TX_CHANNEL_BUSY](#).

In DMP, there's a third error case to handle above the previous two: You should always handle [RAIL_EVENT_SCHEDULER_STATUS](#), where you might receive an error by calling [RAIL_GetSchedulerStatusAlt\(\)](#).

This was implemented because when you call e.g. [RAIL_StartTx\(\)](#), the radio scheduler just creates task. When that task is running, it will actually call the single protocol [RAIL_StartTx\(\)](#), and if that returns an error, it will trigger an [RAIL_EVENT_SCHEDULER_STATUS](#) event.

[RAIL_GetSchedulerStatusAlt\(\)](#) returns two errors:

- [RAIL_SchedulerStatus_t](#)
- [RAIL_Status_t](#)

The first one is difficult to understand if you just read the documentation. It's cleaner if you check it in `rail_types.h`.

The upper 4 bits, masked by `RAIL_SCHEDULER_TASK_MASK` reports the task the error is originated from. For example, `RAIL_SCHEDULER_TASK_SINGLE_TX` means the radio scheduler was working on a [RAIL_StartTx\(\)](#) task.

The lower 4 bits, masked by `RAIL_SCHEDULER_STATUS_MASK` is the error itself, which can have one of the `RAIL_SCHEDULER_STATUS_*` errors (see the [API doc](#) for details).

Usually an application will need to implement `RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL` and `RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED` specifically. These could normally happen, resulted by conflicts with other radio tasks. The first one means the task is impossible to schedule due to other tasks, while the latter means that the task started, but was interrupted by a higher priority task. *Note that these errors happen rarely so one must carefully implement the error handling in a DMP app based on what could theoretically happen.*

The other errors mean more serious problems. `RAIL_SCHEDULER_STATUS_TASK_FAIL` means the internal, single protocol call of the API returned an error. In this case, the second parameter is also meaningful. RAIL saves the returned error into [RAIL_Status_t](#) parameter.

`RAIL_SCHEDULER_STATUS_INTERNAL_ERROR` shouldn't happen, but is theoretically possible, so should be handled. Finally `RAIL_SCHEDULER_STATUS_UNSUPPORTED` is currently only reported if the [RAIL_GetSchedulerStatus\(\)](#) is called with the single protocol RAIL library.

So, for example `RAIL_SCHEDULER_TASK_SCHEDULED_TX | RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED` means that a task, requested by [RAIL_StartScheduledTx\(\)](#) was interrupted by a higher priority operation.

Debugging

When debugging DMP, Rx/Tx PRS channels are still useful. See the [article about debugging](#) for details.

For DMP based code, writing out to a GPIO when a protocol is scheduled/unscheduled is really useful. This can be easily done by setting a GPIO on [RAIL_EVENT_CONFIG_SCHEDULED](#) and clearing it on [RAIL_EVENT_CONFIG_UNCHEDULED](#).

Recommendations and Practices

Although these practices are important for DMP applications, you should consider applying them in single protocol applications as well, to simplify a potential future port to DMP:

- Only use [RAIL_Idle\(\)](#) when it's necessary - In RAIL 1.x almost all APIs required that they be called from idle mode. This requirement was removed in RAIL 2.x, so it's rarely needed, and since [RAIL_Idle\(\)](#) also yields the radio, it should be rarely used in DMP.
- Use the scheduling Rx/Tx features as much as possible - i.e. don't use timers and start Rx/Tx at timer interrupt, use the corresponding RAIL schedule API instead. This helps the radio scheduler to resolve conflicts with the configured `slipTimes`, or at least promptly let the application know about an unavoidable conflict.
- Set the `slipTime` / `transactionTime` correctly. Again, this helps the radio scheduler to resolve conflicts.
- Implement careful error handling in the [RAIL_EVENT_SCHEDULER_STATUS](#) event

How Not to Use DMP

Using multiple DMP instances for multiple radio tasks (e.g., one protocol for advertising, one for connection, or one for Tx, one for Rx) is bad design: while it works, each protocol instance has a significant memory footprint, and switching time between protocols is much slower than switching Rx/Tx inside a protocol. At the moment, it is recommended to use DMP only for serving separate protocol stacks.

Using DMP to handle multiple PHYs for the same protocol stack is also not recommended: Multi-PHY is a much better solution for that, see the [introduction to Multi-PHY and Multiprotocol article](#) for more details.

Related Documentation

- [Dynamic Multiprotocol User's Guide](#)
 - You can skip the chapters that won't affect your design. For example, if you don't plan to use Zigbee, you can skip the Zigbee chapters. This article somewhat overlaps with the Dynamic Multiprotocol User's Guide as we provide a short summary of the basics here.
- [Dynamic Multiprotocol Development with Bluetooth and Proprietary Protocols on RAIL in GSDK v3.x and Higher](#)
 - Read it if you plan to use Bluetooth - RAIL DMP, as it presents how to create Bluetooth - RAIL examples.
- [The Multiprotocol description in the API doc](#) - It is the concentrated description of the Radio Scheduler, and also overlaps with this article.

If you find any conflicts between this article and the above documents, give those documents priority as they will more quickly receive updates to track new RAIL DMP features and guidance.