AN1113: Porting RAIL Applications to RAIL Version 2.x

Silicon Labs RAIL (Radio Abstraction Interface Layer) provides an intuitive, easily-customizable radio interface layer that is designed to support proprietary or standards-based wireless protocols. This application note describes the Application Programming Interface (API) changes from RAIL 1.6 to RAIL 2.x.

KEY POINTS

- Introduces RAIL 2.x.
- Provides lists with comments of the added, removed, unchanged, and modified APIs from RAIL 1.6 to RAIL 2.x.
- Includes sections with the API differences from RAIL 1.6 to RAIL 2.x along with example code for each version of RAIL.

Table of Contents

1. Introduction

Silicon Labs RAIL (Radio Abstraction Interface Layer) is a fundamental building block 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.

As RAIL has evolved and grown, it became apparent that a significant refactoring of the RAIL code was necessary to achieve the longer term goals of RAIL. As a result, Silicon Labs developed RAIL 2.x.

The most significant changes from RAIL 1.6 to RAIL 2.x are as follows:

- Updated APIs to allow for Dynamic Multiprotocol functionality.
- Unified naming across all APIs to follow a strict VerbNoun naming convention.
- A more powerful channel-based approach to configuring the radio. This adds the ability to automatically change radio configurations and the max output power on a per channel basis.
- A unified callback mechanism for all RAIL events.
- A more advanced packet receive API to allow for greater control over when to process incoming packets.

These are the motivations behind RAIL 2.x.

Enable Dynamic Multiprotocol operation:

- Support multiple radio configurations simultaneously on one radio using time slicing.
- Allow multiple instances of RAIL.
- Enable multiple priority levels to support overriding long running but low-priority radio operations with higher priority radio operations from other RAIL instances.
- Permit each transmit and receive radio operation to specify bounds on when it must execute. If it cannot be scheduled within these bounds, it is canceled and reported as dropped to the calling stack.

  Note: RAIL 2.x includes core features for Dynamic Multiprotocol support. Silicon Labs has initially targeted our Zigbee and *Bluetooth* Low Energy (Bluetooth LE) stacks. Support for other protocols and proprietary stacks may be added in future releases.

Improve usability and readability of the code:

- Conform to Silicon Labs coding conventions and APIs.
- Streamline callbacks.
- Reduce the number of APIs.
- Simplify the way TX/RX options are specified.
- Simplify Auto-Ack functionality

Improved functionality and performance:

- Power Amplifier (PA) configurability through RAIL.
- Packet Trace Interface (PTI) configurability through RAIL.
- Support for up to three simultaneous IEEE 802.15.4 Private Area Network (PAN) identifiers, short addresses, and long addresses in the filtering logic on the EFR32 series of chips.

While the RAIL 2.x changes are far-reaching, many of them are largely cosmetic, and none of them should remove any previous functionality. To help migrate your existing RAIL 1.x application to RAIL 2.x there is an example porting layer that can be found at https://www.silabs.com/products/development-tools/software/radio-abstraction-interface-layer-sdk. By including the rail1xx_compatibility.h header file instead of rail.h and building with the rail1xx_compatibility.c file, your RAIL 1.x application should work with the RAIL 2.x library. Keep in mind that this porting layer is provided as an example and should not be used in production code. Silicon Labs has not thoroughly tested or QA'd this porting layer.

2. RAIL 2.x API Overview

The RAIL 2.x Application Programming Interface (API) has changed significantly since RAIL 1.6. The API changes are grouped into these categories:

- Added: new APIs added in RAIL 2.x.
- Removed: APIs in RAIL 1.6 that were deleted in RAIL 2.x.
- Unchanged: APIs that are the same in RAIL 1.6 and RAIL 2.x.
- Modified: APIs in RAIL 1.6 that were changed in RAIL 2.x (this is the majority of the differences).

Consult the *RAIL 2.x API Reference* for complete technical details.

2.1 APIs Added in RAIL 2.x

The following table lists the APIs that were added in RAIL 2.x. The APIs are in alphabetical order. Consult the *RAIL 2.x API Reference* for complete technical details.

Table 2.1. APIs Added in RAIL 2.x

RAIL 2.x API Name	Return Value
RAIL_ConfigSleep(RAIL_Handle_t railHandle, RAIL_SleepConfig_t sleepConfig)	RAIL_Status_t
RAIL_ConvertDbmToRaw(RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPower_t power)	RAIL_TxPowerLevel_t
RAIL_ConvertRawToDbm(RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPowerLevel_t powerLevel)	RAIL_TxPower_t
RAIL_EnableTxHoldOff(RAIL_Handle_t railHandle, bool enable)	void
RAIL_GetActiveChannelConfig(RAIL_Handle_t railHandle)	const RAIL_ChannelConfig_t
RAIL_GetActiveChannelConfigEntry(RAIL_Handle_t railHandle)	const RAIL_ChannelConfigEntry_t
RAIL_GetPtiConfig(RAIL_Handle_t railHandle, RAIL_PtiConfig_t *ptiConfig)	RAIL_Status_t
RAIL_GetRxPacketInfo(RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, RAIL_RxPacketInfo_t *pPacketInfo)	RAIL_RxPacketHandle_t
RAIL_GetSchedulerStatus(RAIL_Handle_t railHandle)	RAIL_SchedulerStatus_t
RAIL_GetTxPacketDetails(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)	RAIL_Status_t
RAIL_GetTxPower(RAIL_Handle_t railHandle)	RAIL_TxPowerLevel_t
RAIL_GetTxPowerConfig(RAIL_Handle_t railHandle, RAIL_TxPowerConfig_t *config)	RAIL_Status_t
RAIL_HoldRxPacket(RAIL_Handle_t railHandle)	RAIL_RxPacketHandle_t
RAIL_IEEE802154_GetAddress(RAIL_Handle_t railHandle, RAIL_IEEE802154_Address_t *pAddress)	RAIL_Status_t
RAIL_IsTxHoldOffEnabled(RAIL_Handle_t railHandle)	bool
RAIL_ReleaseRxPacket(RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle)	RAIL_Status_t
RAIL_SetTxFifo(RAIL_Handle_t railHandle, uint8_t *txBufPtr, uint16_t initLength, uint16_t size)	uint16_t
RAIL_SetTxPower(RAIL_Handle_t railHandle, RAIL_TxPowerLevel_t powerLevel)	RAIL_Status_t
RAIL_Sleep(uint16_t wakeupProcessTime, bool *deepSleepAllowed)	RAIL_Status_t
RAIL_Wake(uint32_t elapsedTime)	RAIL_Status_t
RAIL_YieldRadio(RAIL_Handle_t railHandle)	void

2.2 APIs Removed in RAIL 2.x

The following table summarizes the APIs that were removed in RAIL 2.x. The removed APIs are grouped into these categories:

- Power Amplifier (PA): These are now native RAIL APIs. They have been removed and replaced with a simpler set of APIs for customer use.
- RAIL Callbacks: RAIL 2.x consolidates callbacks into a generic callback with an event mask. As a result, all explicit callback functions have been removed and you must now query any parameters they were being passed in the context of the generic callback.
- Bit Error Rate (BER): This mode no longer needs to be supported with specific APIs. You can do bit error rate testing by properly configuring your PHY and then using the raw bytes from the receive FIFO. For more information, consult the comments in the RAIL 2.x source code.
- Miscellaneous: See the "Comment" column for an explanation about why this API was removed.

Table 2.2. APIs Removed in RAIL 2.x

RAIL 2.x API Name	Return Value	Comment
Power Amplifier (PA) APIs		
`PA_EnableCal(bool enable)`	`void`	PA APIs are now official `RAIL_` APIs.
`PA_MaxOutputPowerSet(void)`	`int32_t`	PA APIs are now official `RAIL_` APIs.
`PA_OutputPowerGet(void)`	`int32_t`	PA APIs are now official `RAIL_` APIs.
`PA_OutputPowerSet(int32_t power)`	`int32_t`	PA APIs are now official `RAIL_` APIs.
`PA_PowerLevelOptimize(int32_t power)`	`void`	PA APIs are now official `RAIL_` APIs.
`PA_PowerLevelSet(uint8_t pwrLevel, uint8_t boostMode)`	`uint16_t`	PA APIs are now official `RAIL_` APIs.
`PA_RampTimeGet(void)`	`uint32_t`	PA APIs are now official `RAIL_` APIs.
`PA_RampTimeSet(uint32_t ramptime)`	`uint32_t`	PA APIs are now official `RAIL_` APIs.
RAIL Callback APIs		
`RAILCb_AllocateMemory(uint32_t size)`	`void *`	All callbacks have been moved to an event in the `RAIL_Config_t.eventsCallback()` function.
`RAILCb_BeginWriteMemory(void *handle, uint32_t offset, uint32_t * available)`	`void *`	All callbacks have been moved to an event in the `RAIL_Config_t.eventsCallback()` function.
`RAILCb_CalNeeded(void)`	`void`	All callbacks have been moved to an event in the `RAIL_Config_t.eventsCallback()` function.
`RAILCb_EndWriteMemory(void *handle, uint32_t offset, uint32_t size)`	`void`	All callbacks have been moved to an event in the `RAIL_Config_t.eventsCallback()` function.
`RAILCb_FreeMemory(void *handle)`	`void`	All callbacks have been moved to an event in the `RAIL_Config_t.eventsCallback()` function.
`RAILCb_IEEE802154_DataRequestCommand(RAIL_IEEE802154_Address_t *address)`	`void`	All callbacks have been moved to an event in the `RAIL_Config_t.eventsCallback()` function.
`RAILCb_RadioStateChanged(uint8_t state)`	`void`	All callbacks have been moved to an event in the `RAIL_Config_t.eventsCallback()` function.
`RAILCb_RfReady(void)`	`void`	All callbacks have been moved to an event in the `RAIL_Config_t.eventsCallback()` function.

RAIL 2.x API Name	Return Value	Comment
RAILCb_RssiAverageDone(int16_t avgRssi)	void	All callbacks have been moved to an event in the RAIL_Config_t.eventsCallback() function.
RAILCb_RxAckTimeout(void)	void	All callbacks have been moved to an event in the RAIL_Config_t.eventsCallback() function.
RAILCb_RxFifoAlmostFull(uint16_t bytesAvailable)	void	All callbacks have been moved to an event in the RAIL_Config_t.eventsCallback() function.
RAILCb_RxPacketReceived(void *rxPacketHandle)	void	All callbacks have been moved to an event in the RAIL_Config_t.eventsCallback() function.
RAILCb_RxRadioStatus(uint8_t status)	void	All callbacks have been moved to an event in the RAIL_Config_t.eventsCallback() function.
RAILCb_RxRadioStatusExt(uint32_t status)	void	All callbacks have been moved to an event in the RAIL_Config_t.eventsCallback() function.
RAILCb_TimerExpired(void)	void	All callbacks have been moved to an event in the RAIL_Config_t.eventsCallback() function.
RAILCb_TxFifoAlmostEmpty(uint16_t spaceAvailable)	void	All callbacks have been moved to an event in the RAIL_Config_t.eventsCallback() function.
RAILCb_TxPacketSent(RAIL_TxPacketInfo_t *txPacketInfo)	void	All callbacks have been moved to an event in the RAIL_Config_t.eventsCallback() function.
RAILCb_TxRadioStatus(uint8_t status)	void	All callbacks have been moved to an event in the RAIL_Config_t.eventsCallback() function.
Bit Error Rates (BER) APIs		
RAIL_BerConfigSet(RAIL_BerConfig_t *berConfig)	void	Bit error rate now uses the normal receive APIs.
RAIL_BerConfigSet(RAIL_BerConfig_t *berConfig)	void	Bit error rate now uses the normal receive APIs.
RAIL_BerRxStart(void)	void	Bit error rate now uses the normal receive APIs.
RAIL_BerRxStop(void)	void	Bit error rate now uses the normal receive APIs.
RAIL_BerStatusGet(RAIL_BerStatus_t *status)	void	Bit error rate now uses the normal receive APIs.
Miscellaneous APIs		
RAIL_AddressFilterByFrameType(uint8_t validFrames)	bool	Moved to an option in the RAIL_FrameType_t configuration.
RAIL_DisableRxFifoThreshold(void)	void	This is now controlled by turning off the RAIL_EVENT_RX_FIFO_ALMOST_FULL event.

RAIL 2.x API Name	Return Value	Comment
`RAIL_EnableRxFifoThreshold(void)`	`void`	This is now controlled by turning on the `RAIL_EVENT_RX_FIFO_ALMOST_FULL` event.
`RAIL_PacketLengthConfigFrameType(const RAIL_FrameType_t *frameType)`	`void`	This API was removed and can now only be set via the radio configuration GUI.
`RAIL_PollAverageRSSI(uint32_t averageTimeUs)`	`int16_t`	This API was removed and should be replaced with the hardware-backed `RAIL_StartAverageRssi()` version.
`RAIL_SetAbortScheduledTxDuringRx(bool abort)`	`void`	Moved to the scheduled transmit configuration.

2.3 APIs Unchanged in RAIL 2.x

The following table lists the APIs that were unchanged from RAIL 1.6 to RAIL 2.x.

Table 2.3. APIs Unchanged in RAIL 2.x

RAIL 2.x API Name	Return Value
`RAIL_EnablePaCal(bool enable)`	`void`
`RAIL_GetTime(void)`	`uint32_t`
`RAIL_SetTime(uint32_t time)`	`RAIL_Status_t`

2.4 APIs Modified in RAIL 2.x

The following table summarizes the APIs that were modified in RAIL 2.x. The list is sorted alphabetically by RAIL 1.6 API name to make it easy for you to find what you need. Many of the APIs were renamed for consistency and had a RAIL handle added to them but the functionality did not fundamentally change.

Table 2.4. APIs Modified from RAIL 1.6 to RAIL 2.x

RAIL 1.6 API Name	Return Value	RAIL 2.x API Name	Return Value	Comment
`RADIO_PA_Init(RADIO_PAInit_t *paInit)`	`bool`	`RAIL_ConfigTxPower(RAIL_Handle_t railHandle, const RAIL_TxPowerConfig_t *config)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RADIO_PTI_Disable(void)`	`void`	`RAIL_EnablePti(RAIL_Handle_t railHandle, bool enable)`	`RAIL_Status_t`	With `enable` = `false`. This API was renamed and given a `RAIL` handle.
`RADIO_PTI_Enable(void)`	`void`	`RAIL_EnablePti(RAIL_Handle_t railHandle, bool enable)`	`RAIL_Status_t`	With `enable` = `true`. This API was renamed and given a `RAIL` handle.
`RADIO_PTI_Init(RADIO_PTIInit_t *ptiInit)`	`void`	`RAIL_ConfigPti(RAIL_Handle_t railHandle, const RAIL_PtiConfig_t *config)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAILCb_AssertFailed(uint32_t errorCode)`	`void`	`RAILCb_AssertFailed(RAIL_Handle_t railHandle, uint32_t errorCode)`	`void`	This API now takes a `RAIL` handle.
`RAIL_AddressFilterConfig(RAIL_AddrConfig_t *addrConfig)`	`bool`	`RAIL_ConfigAddressFilter(RAIL_Handle_t railHandle, const RAIL_AddrConfig_t *addrConfig)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_AddressFilterDisable(void)`	`bool`	`RAIL_EnableAddressFilter(RAIL_Handle_t railHandle, bool enable)`	`bool`	With `enable` = `false`. This API was renamed and given a `RAIL` handle.
`RAIL_AddressFilterDisableAddress(uint8_t field, uint8_t index)`	`bool`	`RAIL_EnableAddressFilterAddress(RAIL_Handle_t railHandle, bool enable, uint8_t field, uint8_t index)`	`RAIL_Status_t`	With `enable` = `false`. This API was renamed and given a `RAIL` handle.
`RAIL_AddressFilterEnable(void)`	`bool`	`RAIL_EnableAddressFilter(RAIL_Handle_t railHandle, bool enable)`	`bool`	With `enable` = `true`. This API was renamed and given a `RAIL` handle.
`RAIL_AddressFilterEnableAddress(uint8_t field, uint8_t index)`	`bool`	`RAIL_EnableAddressFilterAddress(RAIL_Handle_t railHandle, bool enable, uint8_t field, uint8_t index)`	`RAIL_Status_t`	With `enable` = `true`. This API was renamed and given a `RAIL` handle.

RAIL 1.6 API Name	Return Value	RAIL 2.x API Name	Return Value	Comment
`RAIL_AddressFilterIsEnabled(void)`	`bool`	`RAIL_IsAddressFilterEnabled(RAIL_Handle_t railHandle)`	`bool`	This API was renamed and given a `RAIL` handle.
`RAIL_AddressFilterReset(void)`	`void`	`RAIL_ResetAddressFilter(RAIL_Handle_t railHandle)`	`void`	This API was renamed and given a `RAIL` handle.
`RAIL_AddressFilterSetAddress(uint8_t field, uint8_t index, uint8_t *value, bool enable)`	`bool`	`RAIL_SetAddressFilterAddress(RAIL_Handle_t railHandle, uint8_t field, uint8_t index, const uint8_t *value, bool enable)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_AutoAckCancelAck(void)`	`bool`	`RAIL_CancelAutoAck(RAIL_Handle_t railHandle)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_AutoAckConfig(RAIL_AutoAckConfig_t *config)`	`RAIL_Status_t`	`RAIL_ConfigAutoAck(RAIL_Handle_t railHandle, const RAIL_AutoAckConfig_t *config)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_AutoAckDisable(void)`	`RAIL_Status_t`	`RAIL_ConfigAutoAck(RAIL_Handle_t railHandle, const RAIL_AutoAckConfig_t *config)`	`RAIL_Status_t`	With `config.enable` = `false`. This API is now a field in the Auto ACK configuration.
`RAIL_AutoAckIsEnabled(void)`	`bool`	`RAIL_IsAutoAckEnabled(RAIL_Handle_t railHandle)`	`bool`	This API was renamed and given a `RAIL` handle.
`RAIL_AutoAckLoadBuffer(RAIL_AutoAckData_t *ackData)`	`RAIL_Status_t`	`RAIL_WriteAutoAckFifo(RAIL_Handle_t railHandle, const uint8_t *dataPtr, uint8_t dataLength)`	`RAIL_Status_t`	ACK data is split out into explicit parameters like a data pointer and length.
`RAIL_AutoAckRxIsPaused(void)`	`bool`	`RAIL_IsRxAutoAckPaused(RAIL_Handle_t railHandle)`	`bool`	This API was renamed and given a `RAIL` handle.
`RAIL_AutoAckRxPause(void)`	`void`	`RAIL_PauseRxAutoAck(RAIL_Handle_t railHandle, bool pause)`	`void`	With `pause` = `true`. There is now one API for enable and disable that takes a Boolean parameter.
`RAIL_AutoAckRxResume(void)`	`void`	`RAIL_PauseRxAutoAck(RAIL_Handle_t railHandle, bool pause)`	`void`	With `pause` = `false`. There is now one API for enable and disable that takes a Boolean parameter.
`RAIL_AutoAckTxIsPaused(void)`	`bool`	`RAIL_IsTxAutoAckPaused(RAIL_Handle_t railHandle)`	`bool`	This API was renamed and given a `RAIL` handle.

RAIL 1.6 API Name	Return Value	RAIL 2.x API Name	Return Value	Comment
`RAIL_AutoAckTxPause(void)`	`void`	`RAIL_PauseTxAutoAck(RAIL_Handle_t railHandle, bool pause)`	`void`	With `pause` = `true`. There is now one API for enable and disable that takes a Boolean parameter.
`RAIL_AutoAckTxResume(void)`	`void`	`RAIL_PauseTxAutoAck(RAIL_Handle_t railHandle, bool pause)`	`void`	With `pause` = `false`. There is now one API for enable and disable that takes a Boolean parameter.
`RAIL_AutoAckUseTxBuffer(void)`	`bool`	`RAIL_UseTxFifoForAutoAck(RAIL_Handle_t railHandle)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_AutoAckWaitingForAck(void)`	`bool`	`RAIL_IsAutoAckWaitingForAck(RAIL_Handle_t railHandle)`	`bool`	This API was renamed and given a `RAIL` handle.
`RAIL_AverageRSSIReady(void)`	`bool`	`RAIL_IsAverageRssiReady(RAIL_Handle_t railHandle)`	`bool`	This API was renamed and given a `RAIL` handle.
`RAIL_BLE_Deinit(void)`	`void`	`RAIL_BLE_Deinit(RAIL_Handle_t railHandle)`	`void`	This API now takes a `RAIL` handle.
`RAIL_BLE_Init(void)`	`void`	`RAIL_BLE_Init(RAIL_Handle_t railHandle)`	`void`	This API now takes a `RAIL` handle.
`RAIL_BLE_IsEnabled(void)`	`bool`	`RAIL_BLE_IsEnabled(RAIL_Handle_t railHandle)`	`bool`	This API now takes a `RAIL` handle.
`RAIL_BLE_SetPhy1Mbps(void)`	`bool`	`RAIL_BLE_ConfigPhy1Mbps(RAIL_Handle_t railHandle)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_BLE_SetPhy1MbpsViterbi(void)`	`bool`	`RAIL_BLE_ConfigPhy1MbpsViterbi(RAIL_Handle_t railHandle)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_BLE_SetPhy2Mbps(void)`	`bool`	`RAIL_BLE_ConfigPhy2Mbps(RAIL_Handle_t railHandle)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_BLE_SetPhy2MbpsViterbi(void)`	`bool`	`RAIL_BLE_ConfigPhy2MbpsViterbi(RAIL_Handle_t railHandle)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_BLE_SetPhyCoded(RAIL_BLE_Coding_t ble_coding)`	`bool`	`RAIL_BLE_ConfigPhyCoded(RAIL_Handle_t railHandle, RAIL_BLE_Coding_t ble_coding)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.

RAIL 1.6 API Name	Return Value	RAIL 2.x API Name	Return Value	Comment
`RAIL_BLE_SetupChannelRadioParams(uint32_t crcInit, uint32_t accessAddress, uint8_t channel, bool disableWhitening)`	`bool`	`RAIL_BLE_ConfigChannelRadioParams(RAIL_Handle_t railHandle, uint32_t crcInit, uint32_t accessAddress, uint16_t channel, bool disableWhitening)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_BitRateGet(void)`	`uint32_t`	`RAIL_GetBitRate(RAIL_Handle_t railHandle)`	`uint32_t`	This API was renamed and given a `RAIL` handle.
`RAIL_CalInit(const RAIL_CalInit_t *railCalInit)`	`uint8_t`	`RAIL_ConfigCal(RAIL_Handle_t railHandle, RAIL_CalMask_t calEnable)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_CalPendingGet(void)`	`RAIL_CalMask_t`	`RAIL_GetPendingCal(RAIL_Handle_t railHandle)`	`RAIL_CalMask_t`	This API was renamed and given a `RAIL` handle.
`RAIL_CalStart(RAIL_CalValues_t *calValues, RAIL_CalMask_t calForce, bool calSave)`	`void`	`RAIL_Calibrate(RAIL_Handle_t railHandle, RAIL_CalValues_t *calValues, RAIL_CalMask_t calForce)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_CcaCsma(void *params)`	`uint8_t`	`RAIL_StartCcaCsmaTx(RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_CsmaConfig_t *csmaConfig, const RAIL_SchedulerInfo_t *schedulerInfo)`	`RAIL_Status_t`	Instead of being passed as an argument to the `RAIL_TxStart()` function, this is now an independent API that starts a transmit with the given configuration.
`RAIL_CcaLbt(void *params)`	`uint8_t`	`RAIL_StartCcaLbtTx(RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_LbtConfig_t *lbtConfig, const RAIL_SchedulerInfo_t *schedulerInfo)`	`RAIL_Status_t`	Instead of being passed as an argument to the `RAIL_TxStart()` function, this is now an independent API that starts a transmit with the given configuration.
`RAIL_ChannelConfig(const RAIL_ChannelConfig_t *config)`	`uint8_t`	`RAIL_ConfigChannels(RAIL_Handle_t railHandle, const RAIL_ChannelConfig_t *config, RAIL_RadioConfigChangedCallback_t cb)`	`uint16_t`	This API was renamed and given a `RAIL` handle.
`RAIL_ChannelExists(uint8_t channel)`	`RAIL_Status_t`	`RAIL_IsValidChannel(RAIL_Handle_t railHandle, uint16_t channel)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_DataConfig(RAIL_DataConfig_t *dataConfig)`	`RAIL_Status_t`	`RAIL_ConfigData(RAIL_Handle_t railHandle, const RAIL_DataConfig_t *dataConfig)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_DirectModeConfig(bool enable)`	`void`	`RAIL_EnableDirectMode(RAIL_Handle_t railHandle, bool enable)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_GetAverageRSSI(void)`	`int16_t`	`RAIL_GetAverageRssi(RAIL_Handle_t railHandle)`	`int16_t`	This API was renamed and given a `RAIL` handle.

RAIL 1.6 API Name	Return Value	RAIL 2.x API Name	Return Value	Comment
RAIL_GetRadioEntropy(uint8_t *buffer, uint16_t bytes)	uint16_t	RAIL_GetRadioEntropy(RAIL_Handle_t railHandle, uint8_t *buffer, uint16_t bytes)	uint16_t	This API now takes a RAIL handle.
RAIL_GetRxFifoBytesAvailable(void)	uint16_t	RAIL_GetRxFifoBytesAvailable(RAIL_Handle_t railHandle)	uint16_t	This API now takes a RAIL handle.
RAIL_GetRxFifoThreshold(void)	uint16_t	RAIL_GetRxFifoThreshold(RAIL_Handle_t railHandle)	uint16_t	This API now takes a RAIL handle.
RAIL_GetRxFreqOffset(void)	RAIL_FrequencyOffset_t	RAIL_GetRxFreqOffset(RAIL_Handle_t railHandle)	RAIL_FrequencyOffset_t	This API now takes a RAIL handle.
RAIL_GetTune(void)	uint32_t	RAIL_GetTune(RAIL_Handle_t railHandle)	uint32_t	This API now takes a RAIL handle.
RAIL_GetTxFifoSpaceAvailable(void)	uint16_t	RAIL_GetTxFifoSpaceAvailable(RAIL_Handle_t railHandle)	uint16_t	This API now takes a RAIL handle.
RAIL_GetTxFifoThreshold(void)	uint16_t	RAIL_GetTxFifoThreshold(RAIL_Handle_t railHandle)	uint16_t	This API now takes a RAIL handle.
RAIL_IEEE802154_2p4GHzRadioConfig(void)	RAIL_Status_t	RAIL_IEEE802154_Config2p4GHzRadio(RAIL_Handle_t railHandle)	RAIL_Status_t	This API was renamed and given a RAIL handle.
RAIL_IEEE802154_AcceptFrames(uint8_t framesMask)	RAIL_Status_t	RAIL_IEEE802154_AcceptFrames(RAIL_Handle_t railHandle, uint8_t framesMask)	RAIL_Status_t	This API now takes a RAIL handle.
RAIL_IEEE802154_Deinit(void)	RAIL_Status_t	RAIL_IEEE802154_Deinit(RAIL_Handle_t railHandle)	RAIL_Status_t	This API now takes a RAIL handle.
RAIL_IEEE802154_Init(RAIL_IEEE802154_Config_t *config)	RAIL_Status_t	RAIL_IEEE802154_Init(RAIL_Handle_t railHandle, const RAIL_IEEE802154_Config_t *config)	RAIL_Status_t	This API now takes a RAIL handle.
RAIL_IEEE802154_IsEnabled(void)	bool	RAIL_IEEE802154_IsEnabled(RAIL_Handle_t railHandle)	bool	This API now takes a RAIL handle.
RAIL_IEEE802154_SetAddresses(RAIL_IEEE802154_AddrConfig_t *addresses)	bool	RAIL_IEEE802154_SetAddresses(RAIL_Handle_t railHandle, const RAIL_IEEE802154_AddrConfig_t *addresses)	RAIL_Status_t	This API now takes a RAIL handle.
RAIL_IEEE802154_SetFramePending(void)	RAIL_Status_t	RAIL_IEEE802154_SetFramePending(RAIL_Handle_t railHandle)	RAIL_Status_t	This API now takes a RAIL handle.
RAIL_IEEE802154_SetLongAddress(uint8_t *longAddr)	bool	RAIL_IEEE802154_SetLongAddress(RAIL_Handle_t railHandle, const uint8_t *longAddr, uint8_t index)	RAIL_Status_t	This API now takes a RAIL handle.

RAIL 1.6 API Name	Return Value	RAIL 2.x API Name	Return Value	Comment
`RAIL_IEEE802154_SetPanCoordinator(bool isPanCoordinator)`	`RAIL_Status_t`	`RAIL_IEEE802154_SetPanCoordinator(RAIL_Handle_t railHandle, bool isPanCoordinator)`	`RAIL_Status_t`	This API now takes a `RAIL` handle.
`RAIL_IEEE802154_SetPanId(uint16_t panId)`	`bool`	`RAIL_IEEE802154_SetPanId(RAIL_Handle_t railHandle, uint16_t panId, uint8_t index)`	`RAIL_Status_t`	This API now takes a `RAIL` handle.
`RAIL_IEEE802154_SetPromiscuousMode(bool enable)`	`RAIL_Status_t`	`RAIL_IEEE802154_SetPromiscuousMode(RAIL_Handle_t railHandle, bool enable)`	`RAIL_Status_t`	This API now takes a `RAIL` handle.
`RAIL_IEEE802154_SetShortAddress(uint16_t shortAddr)`	`bool`	`RAIL_IEEE802154_SetShortAddress(RAIL_Handle_t railHandle, uint16_t shortAddr, uint8_t index)`	`RAIL_Status_t`	This API now takes a `RAIL` handle.
`RAIL_PaCtuneSet(uint8_t txPaCtuneValue, uint8_t rxPaCtuneValue)`	`RAIL_Status_t`	`RAIL_SetPaCTune(RAIL_Handle_t railHandle, uint8_t txPaCtuneValue, uint8_t rxPaCtuneValue)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_PeekRxPacket(uint8_t *pDst, uint16_t len, uint16_t offset)`	`uint16_t`	`RAIL_PeekRxPacket(RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, uint8_t *pDst, uint16_t len, uint16_t offset)`	`uint16_t`	This function is now also used in the normal Rx flow and takes the new packet handle. See the section on packet receive for more information.
`RAIL_RadioConfig(void *radioConfig)`	`uint8_t`	`RAIL_ConfigRadio(RAIL_Handle_t railHandle, RAIL_RadioConfig_t config)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_ReadRxFifo(uint8_t *dataPtr, uint16_t readLength)`	`uint16_t`	`RAIL_ReadRxFifo(RAIL_Handle_t railHandle, uint8_t *dataPtr, uint16_t readLength)`	`uint16_t`	This API now takes a `RAIL` handle.
`RAIL_ReadRxFifoAppendedInfo(RAIL_AppendedInfo_t *appendedInfo)`	`void`	`RAIL_GetRxPacketDetails(RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, RAIL_RxPacketDetails_t *pPacketDetails)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle.
`RAIL_ResetFifo(bool txFifo, bool rxFifo)`	`void`	`RAIL_ResetFifo(RAIL_Handle_t railHandle, bool txFifo, bool rxFifo)`	`void`	This API now takes a `RAIL` handle.
`RAIL_RfIdle(void)`	`void`	`RAIL_Idle(RAIL_Handle_t railHandle, RAIL_IdleMode_t mode, bool wait)`	`void`	This API has become the same as `RAIL_RfIdleExt` and been renamed. To emulate the old behavior call the API with `(RAIL_IDLE_ABORT, false)`.
`RAIL_RfIdleExt(RAIL_RfIdleMode_t mode, bool wait)`	`void`	`RAIL_Idle(RAIL_Handle_t railHandle, RAIL_IdleMode_t mode, bool wait)`	`void`	This API was renamed and given a `RAIL` handle.

RAIL 1.6 API Name	Return Value	RAIL 2.x API Name	Return Value	Comment
`RAIL_RfInit(const RAIL_Init_t *railInit)`	`uint8_t`	`RAIL_Init(RAIL_Config_t *railCfg, RAIL_InitCompleteCallbackPtr_t cb)`	`RAIL_Handle_t`	This API was renamed and given a `RAIL` handle.
`RAIL_RfSense(RAIL_RfSenseBand_t band, uint32_t senseTime, bool enableCb)`	`uint32_t`	`RAIL_StartRfSense(RAIL_Handle_t railHandle, RAIL_RfSenseBand_t band, uint32_t senseTime, RAIL_RfSense_CallbackPtr_t cb)`	`uint32_t`	This API was renamed and given a `RAIL` handle.
`RAIL_RfSensed(void)`	`bool`	`RAIL_IsRfSensed(RAIL_Handle_t railHandle)`	`bool`	This API was renamed and given a `RAIL` handle.
`RAIL_RfStateGet(void)`	`RAIL_RadioState_t`	`RAIL_GetRadioState(RAIL_Handle_t railHandle)`	`RAIL_RadioState_t`	This API was renamed and given a `RAIL` handle.
`RAIL_RxConfig(uint32_t cbToEnable, bool appendedInfoEnable)`	`uint8_t`	`RAIL_ConfigEvents(RAIL_Handle_t railHandle, RAIL_Events_t mask, RAIL_Events_t events)`	`RAIL_Status_t`	All callbacks are now set in `RAIL_ConfigEvents()` and may have slightly new names. In addition, the `appendedInfoEnable` boolean is now part of the `RxOptions`.
`RAIL_RxGetRSSI(void)`	`int16_t`	`RAIL_GetRssi(RAIL_Handle_t railHandle, bool wait)`	`int16_t`	This API was renamed and given a `RAIL` handle.
`RAIL_RxStart(uint8_t channel)`	`uint8_t`	`RAIL_StartRx(RAIL_Handle_t railHandle, uint16_t channel, const RAIL_SchedulerInfo_t *schedulerInfo)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle and can be passed a schedulerInfo structure for multiprotocol.
`RAIL_ScheduleRx(uint8_t channel, RAIL_ScheduleRxConfig_t *cfg)`	`uint8_t`	`RAIL_ScheduleRx(RAIL_Handle_t railHandle, uint16_t channel, const RAIL_ScheduleRxConfig_t *cfg, const RAIL_SchedulerInfo_t *schedulerInfo)`	`RAIL_Status_t`	This API now takes a `RAIL` handle and can be passed a schedulerInfo structure for multiprotocol.
`RAIL_ScheduleTx(void *params)`	`uint8_t`	`RAIL_StartScheduledTx(RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_ScheduleTxConfig_t *config, const RAIL_SchedulerInfo_t *schedulerInfo)`	`RAIL_Status_t`	Instead of being passed as an argument to the `RAIL_TxStart()` function, this is now an independent API that starts a transmit with the given configuration and can be passed a schedulerInfo structure for multiprotocol.
`RAIL_SetCcaThreshold(int8_t ccaThresholdDbm)`	`RAIL_Status_t`	`RAIL_SetCcaThreshold(RAIL_Handle_t railHandle, int8_t ccaThresholdDbm)`	`RAIL_Status_t`	This API now takes a `RAIL` handle.
`RAIL_SetFixedLength(uint16_t length)`	`uint16_t`	`RAIL_SetFixedLength(RAIL_Handle_t railHandle, uint16_t length)`	`uint16_t`	This API now takes a `RAIL` handle.

RAIL 1.6 API Name	Return Value	RAIL 2.x API Name	Return Value	Comment
RAIL_SetFreqOffset(RAIL_FrequencyOffset_t freqOffset)	RAIL_Status_t	RAIL_SetFreqOffset(RAIL_Handle_t railHandle, RAIL_FrequencyOffset_t freqOffset)	RAIL_Status_t	This API now takes a RAIL handle.
RAIL_SetPtiProtocol(RAIL_PtiProtocol_t protocol)	RAIL_Status_t	RAIL_SetPtiProtocol(RAIL_Handle_t railHandle, RAIL_PtiProtocol_t protocol)	RAIL_Status_t	This API now takes a RAIL handle.
RAIL_SetRxFifoThreshold(uint16_t rxThreshold)	uint16_t	RAIL_SetRxFifoThreshold(RAIL_Handle_t railHandle, uint16_t rxThreshold)	uint16_t	This API now takes a RAIL handle.
RAIL_SetRxOptions(uint32_t options)	RAIL_Status_t	RAIL_ConfigRxOptions(RAIL_Handle_t railHandle, RAIL_RxOptions_t mask, RAIL_RxOptions_t options)	RAIL_Status_t	It is now easier to set individual options and some new values have been added. Check the API documentation for more information.
RAIL_SetRxTransitions(RAIL_RadioState_t success, RAIL_RadioState_t error, uint8_t ignoreErrors)	RAIL_Status_t	RAIL_SetRxTransitions(RAIL_Handle_t railHandle, const RAIL_StateTransitions_t *transitions)	RAIL_Status_t	This API now takes a RAIL handle.
RAIL_SetStateTiming(RAIL_StateTiming_t *timings)	RAIL_Status_t	RAIL_SetStateTiming(RAIL_Handle_t railHandle, RAIL_StateTiming_t *timings)	RAIL_Status_t	This API now takes a RAIL handle.
RAIL_SetTune(uint32_t tune)	void	RAIL_SetTune(RAIL_Handle_t railHandle, uint32_t tune)	RAIL_Status_t	This API now takes a RAIL handle.
RAIL_SetTxFifoThreshold(uint16_t txThreshold)	uint16_t	RAIL_SetTxFifoThreshold(RAIL_Handle_t railHandle, uint16_t txThreshold)	uint16_t	This API now takes a RAIL handle.
RAIL_SetTxTransitions(RAIL_RadioState_t success, RAIL_RadioState_t error)	RAIL_Status_t	RAIL_SetTxTransitions(RAIL_Handle_t railHandle, const RAIL_StateTransitions_t *transitions)	RAIL_Status_t	This API now takes a RAIL handle.
RAIL_StartAverageRSSI(uint8_t channel, uint32_t averagingTimeUs)	RAIL_Status_t	RAIL_StartAverageRssi(RAIL_Handle_t railHandle, uint16_t channel, uint32_t averagingTimeUs, const RAIL_SchedulerInfo_t *schedulerInfo)	RAIL_Status_t	This API was renamed and given a RAIL handle and can be passed a schedulerInfo structure for multiprotocol.
RAIL_SymbolRateGet(void)	uint32_t	RAIL_GetSymbolRate(RAIL_Handle_t railHandle)	uint32_t	This API was renamed and given a RAIL handle.
RAIL_TimerCancel(void)	void	RAIL_CancelTimer(RAIL_Handle_t railHandle)	void	This API was renamed and given a RAIL handle.
RAIL_TimerExpired(void)	bool	RAIL_IsTimerExpired(RAIL_Handle_t railHandle)	bool	This API was renamed and given a RAIL handle.

RAIL 1.6 API Name	Return Value	RAIL 2.x API Name	Return Value	Comment
`RAIL_TimerGet(void)`	`uint32_t`	`RAIL_GetTimer(RAIL_Handle_t railHandle)`	`uint32_t`	This API was renamed and given a `RAIL` handle.
`RAIL_TimerIsRunning(void)`	`bool`	`RAIL_IsTimerRunning(RAIL_Handle_t railHandle)`	`bool`	This API was renamed and given a `RAIL` handle.
`RAIL_TimerSet(uint32_t time, RAIL_TimeMode_t mode)`	`RAIL_Status_t`	`RAIL_SetTimer(RAIL_Handle_t railHandle, uint32_t time, RAIL_TimeMode_t mode, RAIL_TimerCallback_t cb)`	`RAIL_Status_t`	This API was renamed and given a `RAIL` handle and its own callback rather than an event.
`RAIL_TxConfig(uint32_t cbToEnable)`	`RAIL_Status_t`	`RAIL_ConfigEvents(RAIL_Handle_t railHandle, RAIL_Events_t mask, RAIL_Events_t events)`	`RAIL_Status_t`	All callbacks are now set in `RAIL_ConfigEvents()` and may have slightly new names.
`RAIL_TxDataLoad(RAIL_TxData_t *txData)`	`uint8_t`	`RAIL_WriteTxFifo(RAIL_Handle_t railHandle, const uint8_t *dataPtr, uint16_t writeLength, bool reset)`	`uint16_t`	All data loads use the `RAIL_WriteTxFifo()` API. To emulate the old `RAIL_TxDataLoad()` behavior, set the reset parameter to true.
`RAIL_TxPowerGet(void)`	`int32_t`	`RAIL_GetTxPowerDbm(RAIL_Handle_t railHandle)`	`RAIL_TxPower_t`	This API was renamed and given a RAIL handle. You may optionally now use power levels instead of dBm as well. See PA changes for more details.
`RAIL_TxPowerSet(int32_t powerLevel)`	`int32_t`	`RAIL_SetTxPowerDbm(RAIL_Handle_t railHandle, RAIL_TxPower_t power)`	`RAIL_Status_t`	This API was renamed and given a RAIL handle. You may optionally now use power levels instead of dBm as well. See PA changes for more details.
`RAIL_TxStart(uint8_t channel, RAIL_PreTxOp_t preTxOp, void *preTxOpParams)`	`uint8_t`	`RAIL_StartTx(RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_SchedulerInfo_t *schedulerInfo)`	`RAIL_Status_t`	This API no longer takes a `preTxOp` so it only does immediate transmits. In addition, it requires `TxOptions` and can be passed a schedulerInfo structure for multiprotocol.
`RAIL_TxStartWithOptions(uint8_t channel, RAIL_TxOptions_t *options, RAIL_PreTxOp_t preTxOp, void *preTxOpParams)`	`uint8_t`	`RAIL_StartTx(RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_SchedulerInfo_t *schedulerInfo)`	`RAIL_Status_t`	This API has been replaced by `RAIL_StartTx()` because it always takes `TxOptions`.

RAIL 1.6 API Name	Return Value	RAIL 2.x API Name	Return Value	Comment
RAIL_TxStreamStart(uint8_t channel, RAIL_StreamMode_t mode)	uint8_t	RAIL_StartTxStream(RAIL_Handle_t railHandle, uint16_t channel, RAIL_StreamMode_t mode)	RAIL_Status_t	This API was renamed, given a RAIL handle, and now supports emitting a tone through the mode parameter.
RAIL_TxStreamStop(void)	uint8_t	RAIL_StopTxStream(RAIL_Handle_t railHandle)	RAIL_Status_t	This API was renamed and given a RAIL handle.
RAIL_TxToneStart(uint8_t channel)	uint8_t	RAIL_StartTxStream(RAIL_Handle_t railHandle, uint16_t channel, RAIL_StreamMode_t mode)	RAIL_Status_t	A tone can now be output by passing RAIL_STREAM_CARRIER_WAVE to the RAIL_StartTxStream() API.
RAIL_TxToneStop(void)	uint8_t	RAIL_StopTxStream(RAIL_Handle_t railHandle)	RAIL_Status_t	This API is now stopped with the RAIL_StopTxStream() API since they were joined together.
RAIL_VersionGet(RAIL_Version_t *version, bool verbose)	void	RAIL_GetVersion(RAIL_Version_t *version, bool verbose)	void	This API was renamed.
RAIL_WriteTxFifo(uint8_t *dataPtr, uint16_t writeLength)	uint16_t	RAIL_WriteTxFifo(RAIL_Handle_t railHandle, const uint8_t *dataPtr, uint16_t writeLength, bool reset)	uint16_t	This API has a new parameter to control whether the FIFO is reset. To emulate the old behavior, set this parameter to false.

3. Initialization

The following table summarizes the initialization differences between RAIL 1.6 and RAIL 2.x.

Table 3.1. Initialization Differences between RAIL 1.6 and RAIL 2.x

RAIL 1.6	RAIL 2.x
1. `RAIL_RfInit()` was used to initialize RAIL. `RAIL_RfInit()` returned an error code with the status of the RAIL initialization.	1. `RAIL_Init()` returns a `RAIL_Handle_t`. This handle is a reference to an instance of RAIL and is used to perform operations on a given RAIL instance. `RAIL_Init()` returns NULL if `RAIL_Init()` was unsuccessful.
2. `RAIL_RfInit()` took in a `RAIL_Init_t` configuration structure to configure RAIL.	2. The `RAIL_Config_t` argument of `RAIL_Init()` contains configuration and state data for each instance of RAIL. For this reason, a given `RAIL_Config_t` instance must only be used to create a single RAIL instance. `RAIL_Config_t` contains the following fields: 1. The `eventsCallback` field which is a pointer to the one event callback function for all RAIL events. In RAIL 1.6, there were separate callback functions. For more information, see 9. Event Coalescing. 2. The RAIL scheduler requires a `RAILSched_Config_t` structure. The `RAILSched_Config_t` structure is a zero-initialized data structure used by the scheduler. This structure should not be modified by the user after RAIL initialization. If not using the multiprotocol RAIL library, pass a NULL pointer for this field. 3. The `protocol` field contains protocol-specific configuration data. The `protocol` field must be NULL for the IEEE802.15.4 protocol. When initializing Bluetooth low energy, the `protocol` field must point to an instance of `RAIL_BLE_State_t`.
2. The `calEnable` field of `RAIL_Init_t` was used to configure RAIL calibrations. `RAIL_Init_t` also contained fields for crystal frequency and maximum packet length.	2. `RAIL_ConfigCal()` is used to configure calibration. The maximum packet size and crystal frequency are no longer needed as part of the RAIL configuration. On EFR32, RAIL currently supports only 38.4 MHz crystals.
3. `RAILCb_RfReady()` was a user-implemented callback function that was called by RAIL after `RAIL_RfInit()` was completed.	3. `RAIL_Init()` takes a `RAIL_InitCompleteCallbackPtr_t` argument which is called when the initialization of a RAIL instance is completed. Pass NULL if this callback is unneeded.
4. `RAIL_RadioConfig()` was called as part of initialization to load a static radio configuration.	4. `RAIL_ConfigRadio()` does not need to be called as part of RAIL initialization. The configurations are now loaded on a per channel basis by `RAIL_ConfigChannels()`.
5. The user did not need to allocate memory for the transmit FIFO during initialization.	5. The user must allocate memory for the transmit FIFO and call `RAIL_SetTxFifo()` to tell RAIL about it.

3.1 Example Code: RAIL 1.6

A RAIL 1.6 application's initialization code might have looked like this:

```
static const RAIL_Init_t railInitParams = {
  APP_MAX_PACKET_LENGTH,  // UNUSED - Will be removed in a future RAIL version
  RAIL_RF_XTAL_FREQ,      // in Hz, so this is 38400000 on the EFR32
};

static const RAIL_CalInit_t railCalInitParams = {
  RAIL_CAL_ALL,           // Run all possible calibrations for this chip
  irCalConfig,            // Settings specific to IR calibration
};

// Initialize the radio out of startup so that it's ready to receive
int radioInitialize(void)
{
  // Initialize the RAIL library and any internal state it requires
  RAIL_RfInit(&railInitParams);

  // Initialize calibration settings
  RAIL_CalInit(&railCalInitParams);

  // Apply the selected RADIO configuration
  if (RAIL_RadioConfig((void*)configList[0])) {
    // Error: Could not apply the radio configuration
    while(1);
  }

  // Set us to a valid channel for this config
  RAIL_ChannelConfig(channelConfigs[0]);

  // Configure a few error callbacks and enable appended info in the returned
  // output structure
  RAIL_RxConfig((  RAIL_RX_CONFIG_INVALID_CRC
                 | RAIL_RX_CONFIG_ADDRESS_FILTERED),
                true);

  // Set automatic transitions to always receive once started
  RAIL_SetRxTransitions(RAIL_RF_STATE_RX, RAIL_RF_STATE_RX,
                        RAIL_IGNORE_NO_ERRORS);
  RAIL_SetTxTransitions(RAIL_RF_STATE_RX, RAIL_RF_STATE_RX);
}
```

3.2 Example Code: RAIL 2.x

A RAIL 2.x application's initialization code example would now look something like this:

```
#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);

  // We must reapply the Tx power 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
}

static RAIL_Config_t railCfg = { // Must never be const
  .eventsCallback = &radioEventHandler,
  .protocol = NULL, // For BLE, pointer to a RAIL_BLE_State_t
  .scheduler = NULL, // For MultiProtocol, pointer to a RAIL_SchedConfig_t
};

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

  // Configure calibration settings
  RAIL_ConfigCal(gRailHandle, RAIL_CAL_ALL);

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

  // Configure the most useful callbacks plus catch 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);

  // Set 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);

  // Setup the transmit buffer
  RAIL_SetTxFifo(gRailHandle, txFifo, 0, TX_FIFO_SIZE);
}
```

4. Consolidating Radio Configurations

The following table summarizes the radio configuration differences between RAIL 1.6 and RAIL 2.x.

Table 4.1. Radio Configuration Differences between RAIL 1.6 and RAIL 2.x

RAIL 1.6	RAIL 2.x
1. The configuration of the radio was radio-centric in that radio settings were managed independently from other settings (for example, channel, frame, and calibration). It was up to the application to correctly apply the corresponding radio, channel, frame, and calibration settings.	1. The configuration of the radio is channel-centric in that one structure describing channel usage is provided during initialization that contains pointers to other configuration structures necessary for each channel's use. The application can focus on simply using channel numbers; RAIL automatically switches between radio, frame, calibration, and other settings.
2. `RAIL_RadioConfig()` was used to supply a uint32_t array to RAIL for radio configuration purposes.	2. `RAIL_RadioConfig()` is an advanced function that should not normally be used. It should only be used if a non-default radio configuration needs to be applied to the radio hardware. Default radio configurations are pointed to by elements inside the `RAIL_ChannelConfig_t` structure which is passed as an input into `RAIL_ConfigChannels()`.
3. `RAIL_ChannelConfig()` was used to supply a `RAIL_ChannelConfig_t` pointer to RAIL for channel configuration purposes.	3. `RAIL_ConfigChannels()` is used to supply a `RAIL_ChannelConfig_t` pointer to RAIL for use with all designated channels. This channel configuration is associated with the supplied RAIL instance `RAIL_Handle_t`. The `RAIL_ChannelConfig_t` structure, organized according to various channel ranges, includes pointers to the associated radio, frame, calibration, and timing structures necessary for each channel's operation. `RAIL_ChannelConfigEntry_t` points to one non-const structure: `RAIL_ChannelConfigEntryAttr_t`. This structure is used to save calibration values in memory so they can be auto-applied when switching among channel config entries.
4. `RAIL_RadioConfig()`, `RAIL_ChannelConfig()`, and `RAIL_PacketLengthConfigFrameType()` each needed to be called by the application. It was the responsibility of the application to manage each of these APIs to ensure that the correct radio, channel, and frame settings were applied at the appropriate time.	4. Each time a channel is requested for use (for example, `RAIL_StartTx()` or `RAIL_StartRx()`), the RAIL library searches through the `RAIL_ChannelConfigEntry_t` array referenced inside of the `RAIL_ChannelConfig_t` structure in search of the first channel configuration entry that matches the requested channel and current power level. When a channel has been requested for use and no channel config entries are eligible for use due to the current TX power being greater than the maximum TX power specified in the corresponding channel config entry, the radio's TX power will be reduced to the maximum TX power specified in the corresponding channel config entry.
5. `RAIL_PacketLengthConfigFrameType()` was used to supply a `RAIL_FrameType_t` pointer to RAIL for frame configuration purposes.	5. `RAIL_PacketLengthConfigFrameType()` has been deprecated from the RAIL API because this functionality is now automatically performed by the RAIL library.
6. No callback was needed for radio configuration changes because they were applied only by the application.	6. `RAIL_ConfigChannels()` takes as an input a `RAIL_RadioConfigChangedCallback_t` callback pointer. If application-specific code needs to run based upon which channel config entry is in use (for example, switching among hardware PAs), this code may be placed in this callback.

4.1 Example Code: RAIL 1.6

A few examples of different channel configurations in RAIL 1.6:

```
// Ten channels starting a 915 Mhz with a channel spacing of 1 Mhz
RAIL_ChannelConfigEntry_t channels = {
  0, 9, 1000000, 915000000
};
RAIL_ChannelConfig_t channelScheme = {
  channels,
  1
};

// 120 channels starting at 915Mhz with channel spacing of 100KHz
RAIL_ChannelConfigEntry_t channels[] = {
  {0, 63, 100000, 910000000},
  {64, 119, 100000, 916400000},
};
RAIL_ChannelConfig_t channelScheme = {
  channels,
  2
};

// 5 nonlinear channels
RAIL_ChannelConfigEntry_t channels[] = {
  {0, 0, 0, 910123456},
  {1, 1, 0, 911654789},
  {2, 2, 0, 912321456},
  {3, 3, 0, 913147852},
  {4, 4, 0, 914567890}
};
RAIL_ChannelConfig_t channelScheme = {
  channels,
  5
};
```

4.2 Example Code: RAIL 2.x

A few examples of different channel configurations in RAIL 2.x:

```
// 21 channels starting at 2.45GHz with channel spacing of 1MHz
// ... 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 config for this entry
    .baseFrequency = 2450000000,
    .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 config for all entries
  .phyConfigDeltaSubtract = NULL, // Subtract this to restore default config
  .configs = generated_channels,
  .length = 1 // There are this many channel config entries
};
const RAIL_ChannelConfig_t *channelConfigs[] = {
  &generated_channelConfig,
  NULL
};
// ... in main code ...
// Associate a specific channel config 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 config 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, // We want ch 2 = baseFrequency
    .channelNumberEnd = 2,
    .maxPower = RAIL_TX_POWER_MAX,
    .attr = &generated_entryAttr
  },
  {
    .phyConfigDeltaAdd = NULL,
    .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 config for all entries
  .phyConfigDeltaSubtract = NULL, // Subtract this to restore default config
  .configs = generated_channels,
  .length = 4 // There are this many channel config entries
};
const RAIL_ChannelConfig_t *channelConfigs[] = {
  &generated_channelConfig,
  NULL
};
// ... in main code ...
// Associate a specific channel config 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 default config 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 config for all entries
  .phyConfigDeltaSubtract = NULL, // Subtract this to restore default config
  .configs = generated0_channels,
  .length = 1 // There are this many channel config 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 config.
railHandle0 = RAIL_Init(&railCfg0, &RAILCb_RfReady0);
railHandle1 = RAIL_Init(&railCfg1, &RAILCb_RfReady1);
railHandle2 = RAIL_Init(&railCfg2, &RAILCb_RfReady2);
// Associate each channel config 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 config entry.
RAIL_SetTxPowerDbm(railHandle1, 100); // set 10.0 dBm TX power
RAIL_StartRx(railHandle1, 0, &schedInfo); // RX using generated1_channels[2]
RAIL_SetTxPowerDbm(railHandle1, 0); // set 0 dBm TX power
RAIL_StartRx(railHandle1, 0, &schedInfo); // RX using generated1_channels[1]
RAIL_StartRx(railHandle2, 0, &schedInfo); // RX using generated2_channels[0]
```

5. Calibration

The following table summarizes the calibration differences between RAIL 1.6 and RAIL 2.x.

Table 5.1. Calibration Differences between RAIL 1.6 and RAIL 2.x

RAIL 1.6	RAIL 2.x
1. `RAIL_CalInit()` accepted the structure `RAIL_CalInit_t`, which contains the element `calEnable` used to specify which calibrations are to run. `RAIL_RfInit()` with its input structure of `RAIL_Init_t::calEnable` was the original method through which RAIL calibrations were enabled. The configuration of calibrations was eventually moved from `RAIL_Init_t` to `RAIL_CalInit_t`, but the `RAIL_Init_t` structure remained the same for backward compatibility. If `RAIL_Init_t::calEnable` and `RAIL_CalInit_t::calEnable` both existed, the `RAIL_Init_t::calEnable` element was ignored.	1. `RAIL_ConfigCal()` uses the input parameter of `RAIL_CalMask_t` in order to specify those calibrations which are to be enabled.
2. `RAIL_CalInit_t::irCalSettings` provided a means whereby Image Rejection (IR) calibration initialization settings were provided to hardware. It was up to the application to call `RAIL_CalInit()` each time the radio configuration changed in order to apply the correct IR calibration settings for the radio configuration in use.	2. The correct IR calibration initialization settings are applied automatically by the new channel configuration architecture.
3. For calibrations enabled by `RAIL_CalInit::calEnable`, the `RAILCb_CalNeeded()` callback was enabled for application notification when calibration action is needed.	3. For calibrations enabled by the `RAIL_CalMask_t` input to `RAIL_ConfigCal()`, the `RAIL_Config_t::eventsCallback()` is called with a `RAIL_Events_t::RAIL_EVENT_CAL_NEEDED` notification when calibration action is needed.
4. `RAIL_CalStart()` contained a boolean `calSave` input parameter. When this was `true` and specific calibrations were run, the `RAIL_CalValues_t::calValues` structure elements corresponding to the value `RAIL_CAL_INVALID_VALUE` were updated to the new calibration results. If the boolean `calSave` input parameter if `false`, the valid calibration values inside of `RAIL_CalValues_t::calValues`, designated to run according to the input `RAIL_CalMask_t::calForce`, were applied without those calibrations needing to have run.	4. `RAIL_Calibrate()` no longer contains a boolean `calSave` input parameter. If a `RAIL_CalValues_t` structure is provided and the calibration to be performed correlates to an element inside that structure containing the value `RAIL_CAL_INVALID_VALUE`, that invalid value will be updated to a valid calibration value after calibration occurs. If that structure's element already contained a valid calibration value before `RAIL_Calibrate()` is called, that original valid calibration value will be applied instead of invoking that particular calibration algorithm. Once a valid `RAIL_CalValues_t` value is associated with a particular channel configuration entry, that value is cached internally within the `RAIL_ChannelConfig_t::RAIL_ChannelConfigEntry_t::RAIL_ChannelConfigEntryAttr_t::calValues` element. This value is automatically applied when that particular channel configuration entry is used and a calibration event will not be issued to the application. Because this cached value exists in memory, a reset will return the cached calibration value back to an invalid state whereupon the first use of the channel configuration entry will cause a `RAIL_Events_t::RAIL_EVENT_CAL_NEEDED` notification to be issued by RAIL's `RAIL_Config_t::eventsCallback()`.

5.1 Example Code: RAIL 1.6

This example shows how to simply set up and service all RAIL calibration needs. No attempt is made to save calibration results at the application level.

```
// Initialization Information
const RAIL_CalInit_t railCalInitParams = {
  RAIL_CAL_ALL,
  irCalConfig // byte array provided in rail_config.c from radio configurator
};

// Initialize Radio Calibrations (in main code)
RAIL_CalInit(&railCalInitParams);

// Service RAIL Calibration Callback
void RAILCb_CalNeeded()
{
  RAIL_CalStart(NULL, RAIL_CAL_ALL_PENDING, false);
}
```

5.2 Example Code: RAIL 2.x

This example shows how to simply set up and service all RAIL calibration needs. No attempt is made to save calibration results at the application level.

```
// Register an application-level callback for RAIL events
RAIL_Config_t railCfg = {
  .eventsCallback = &RAILCb_Event
};

// Initialize Radio and Register RAIL Events Callback (in main code)
railHandle = RAIL_Init(&railCfg, NULL);

// Initialize Radio Calibrations (in main code)
RAIL_ConfigCal(railHandle, RAIL_CAL_ALL);

// Service RAIL Events Callback
void RAILCb_Event(RAIL_Handle_t railHandle, RAIL_Events_t events)
{
  // Handle only calibration needs here.
  if (events & RAIL_EVENT_CAL_NEEDED) {
    RAIL_Calibrate(railHandle, NULL, RAIL_CAL_ALL_PENDING);
  }
}
```

6. Packet Trace Interface APIs

The following table summarizes the Packet Trace Interface (PTI) API differences between RAIL 1.6 and RAIL 2.x.

Table 6.1. PTI API Differences between RAIL 1.6 and RAIL 2.x

RAIL 1.6	RAIL 2.x
1. `RADIO_PTIInit_t` was the structure passed into `RADIO_PTI_Init()` to configure PTI. Other `RADIO_PTI_...` functions were used to enable the PTI.	1. `RAIL_PtiConfig_t` is the new structure passed into `RAIL_ConfigPti()` to configure PTI. All the `RADIO_PTI_...` functions are now `RAIL_...` functions. RAIL 2.x users should no longer call `RADIO_PTI_...` functions. **Note:** PTI in RAIL 2.x is configured and enabled/disabled for all protocols on a single radio. A `RAIL_Handle_t` is still taken into these functions to support multiple radios in the future. For RAIL 2.x, Silicon Labs recommends always passing in `RAIL_EFR32_HANDLE` (defined in `rail_chip_specific.h`).

6.1 Example Code: RAIL 1.6

The following code outlines PTI configuration and enabling in RAIL 1.6.

```
RADIO_PTIInit_t ptiInit = {
  RADIO_PTI_MODE_UART,    // Simplest output mode is UART mode
  1600000,                // Choose 1.6 MHz for best compatibility
  6,                      // WSTK uses location 6 for DOUT
  gpioPortB,              // Get the port for this loc
  12,                     // Get the pin, location should match above
  0,                      // DCLK not used for UART mode
  gpioPortB,              // DCLK not used for UART mode
  0,                      // DCLK not used for UART mode
  6,                      // WSTK uses location 6 for DFRAME
  gpioPortB,              // Get the port for this loc
  13,                     // Get the pin, location should match above
};

RADIO_PTI_Init(&ptiInit);

// Then, PTI was enabled with
RADIO_PTI_Enable();

// And later, it could be disable with
RADIO_PTI_Disable();
```

6.2 Example Code: RAIL 2.x

The RAIL 2.x version of the code changed very little. The only differences are naming and now taking a `RAIL_Handle_t` parameter (not currently used, but may be in future releases).

```
RAIL_PtiConfig_t ptiConfig = {
  RAIL_PTI_MODE_UART,     // Simplest output mode is UART mode
  1600000,                // Choose 1.6 MHz for best compatibility
  6,                      // WSTK uses location 6 for DOUT
  gpioPortB,              // Get the port for this loc
  12,                     // Get the pin, location should match above
  0,                      // DCLK not used for UART mode
  gpioPortB,              // DCLK not used for UART mode
  0,                      // DCLK not used for UART mode
  6,                      // WSTK uses location 6 for DFRAME
  gpioPortB,              // Get the port for this loc
  13,                     // Get the pin, location should match above
};

// Although not currently used, we recommended passing RAIL_EFR32_HANDLE as the
// RAIL_Handle_t for best future-compatibility
RAIL_ConfigPti(RAIL_EFR32_HANDLE, &ptiConfig);

// Then, PTI is enabled with
RAIL_EnablePti(RAIL_EFR32_HANDLE, true);

// And later, it can be disable with
RAIL_EnablePti(RAIL_EFR32_HANDLE, false);
```

7. Power Amplifier APIs

The following table summarizes the Power Amplifier (PA) API differences between RAIL 1.6 and RAIL 2.x.

Table 7.1. PA API Differences between RAIL 1.6 and RAIL 2.x

RAIL 1.6	RAIL 2.x
1. PA power conversion curves (mapping between raw power levels written to registers ("raw" power or "power level") and dBm (simply "power" or "(actual) output power") were compiled into RAIL library code. If customer boards caused the power curve mappings to not be accurate, there was very little the customer could do to update that relationship, except change the `offset` parameter of `RADIO_PAInit_t`.	1. All of Silicon Labs conversion functions are now completely open source, but default implementations are still built into the library as `WEAK`. There are a few levels of customization available: 1. **No customization**: The power curves provided by Silicon Labs work with the customer application/board. `RAIL_ConvertRawToDbm()` and `RAIL_ConvertDbmToRaw()` can be called directly as compiled into the library. 2. **Curve customization**: The Silicon Labs functions' methodologies for computing the power level to dBm power has sufficient resolution for the customer's needs, but the actual conversion is incorrect for the customer's board. Using `pa_customer_curve_fits.py`, customers can update the curve defines found in `pa_curves_efr32.h` or create their own new file similar to `pa_curves_efr32.h` to define their own curves. 3. **Function customization**: There may be customers who need higher resolution than the Silicon Labs curve-fitting methodology provides, or do not have enough code space to spare for the Silicon Labs functions. In that case, they can completely change the methodology however they like (for example, using a lookup table for specific values). As long as customers override `RAIL_ConvertRawToDbm()` and `RAIL_ConvertDbmToRaw()` with the same signature as used in `rail.h`, they can handle conversions in whatever manner they choose. `RAIL_GetTxPowerCurve()` and `RAIL_InitTxPowerCurves()` are only needed for the Silicon Labs methodology. Depending on the customer's new methods, they may not be needed. These last two functions are not called in the RAIL library, so they can also be removed entirely. 4. **Not needed at all**: Customers may have a protocol/application in which knowing the exact dBm power is not necessary. In this case, Silicon Labs recommends overriding `RAIL_ConvertRawToDbm()` and `RAIL_ConvertDbmToRaw()` with smaller functions (`RAIL_ConvertRawToDbm()` should return `RAIL_TX_POWER_MIN` and `RAIL_ConvertDbmToRaw()` should return 255 to prevent channel power limit coercion). Because the library will call these functions, if customers want to reduce code size, they **must** be overridden with some smaller function. Simply not calling these functions explicitly in the customer application is not sufficient to prevent these (rather large) functions from increasing code space.
2. All APIs that set the power accepted and returned all power values in deci-dBm units (that is, dBm * 10).	2. `RAIL_SetTxPower()` and `RAIL_GetTxPower()` set and return values in power levels (0-252 for the High-Power 2.4GHz PA, 0-7 for the Low-Power 2.4GHz PA, and 0-248 for the subGHz PA). To use deci-dBm values, customers can call `RAIL_ConvertRawToDbm()` or `RAIL_ConvertDbmToRaw()` endpoints, or use `RAIL_SetTxPowerDbm()` and `RAIL_GetTxPowerDbm()`.
3. `RADIO_PAInit_t` was the structure passed into `RADIO_PA_Init()` to configure the PA. It used to contain an offset field that was a best-effort attempt to make up for the fact that curves could not be fully customized. The power (in deci-dBm) was also part of this structure and the `voltMode` field was simply an enum to indicate whether a battery or the DCDC converter was powering the PA (this information is critical because it is used internally in the library to adjust protection circuitry).	3. `RADIO_PAInit_t` has now become `RAIL_TxPowerConfig_t` and is passed into `RAIL_ConfigTxPower()`. The structure no longer contains an offset because curves are now fully open source and configurable. Power is now expressed in power level (not deci-dBm) and is no longer part of this config structure/function. Because it is expected to change more frequently than the rest of the PA configurations, power now has its own `RAIL_SetTxPower()` function. PA `voltage` is now a `uint16_t` to express the PA voltage in terms of mV (millivolts). This value is still critical to ensure that the library correctly sets PA protection circuitry. There should now no longer be any `RADIO_PA_...` functions in customer code.

7.1 Example Code: RAIL 1.6

The PA interface was minimal; there was no need to set up conversion curves. Although this was simpler, it commonly caused dBm powers to be incorrect on customer boards as PA performance was highly dependent on the board on which it is placed.

```
RADIO_PAInit_t paInit = {
  PA_SEL_2P4_HP,    // Power Amplifier mode
  PA_VOLTMODE_DCDC, // Power Amplifier vPA Voltage mode
  100,              // Desired output power in dBm * 10
  0,                // Output power offset in dBm * 10
  10,               // Desired ramp time in us
};
RADIO_PA_Init(&paInit);
```

7.2 Example Code: RAIL 2.x

The PA curves that map raw TX power levels written to the PA registers to actual dBm powers must also be loaded, giving more flexibility to the customer to override them.

```
// First declare the variables required by the curve structure
RAIL_DECLARE_TX_POWER_VBAT_CURVES(piecewiseSegments, curvesSg, curves24Hp, curves24Lp);

// Then, load those variables into the curve structure
RAIL_TxPowerCurvesConfig_t txPowerCurvesConfig = {
  curves24Hp,
  curvesSg,
  curves24Lp,
  piecewiseSegments
};

// And then pass them to the plugin to save the data
RAIL_InitTxPowerCurves(&txPowerCurvesConfig);

// Create the configuration structure, which no longer contains ‘offset’ or ‘power’
// Voltage is specified as an integer, not an enum
RAIL_TxPowerConfig txPowerConfig = {
  RAIL_TX_POWER_MODE_2P4_HP,
  3300,
  100
};

// And pass that structure to the configuration API
RAIL_ConfigTxPower(railHandle, &txPowerConfig);

// Set the power to 100 deci-dBm (10 dBm)
RAIL_SetTxPowerDbm(railHandle, 100);
```

8. AutoACK API Consolidation

The following table summarizes the AutoACK differences between RAIL 1.6 and RAIL 2.X.

Table 8.1. AutoACK Differences between RAIL 1.6 and RAIL 2.X

RAIL 1.6	RAIL 2.X
1. `RAIL_AutoAckConfig()` always configured ACKing **and** turned auto-ACKing on; `RAIL_AutoAckDisable()` always turned it off.	1. Turning auto-ACK on or off are both controlled using the `RAIL_ConfigAutoAck()` API. Turning it on or off is controlled by the `enable` field in `RAIL_AutoAckConfig_t`.
2. State timing was part of the `RAIL_AutoAckConfig_t` structure. It was set up as part of `RAIL_AutoAckConfig()` and was not reverted to pre-ACKing timings in `RAIL_AutoAckDisable()`.	2. State timing is now not touched at all in `RAIL_ConfigAutoAck()`. So you must call `RAIL_SetStateTiming()` explicitly if you want to update timings. There is one caveat: While auto-ACK is enabled, `txToRx` of `RAIL_StateTiming_t` should be set to 10 less than desired to make absolutely sure that the radio can get to RX in time to receive an ACK when it is expecting one.
3. After calling `RAIL_AutoAckConfig()`, the radio would enter the state specified in `RAIL_AutoAckConfig_t`'s `defaultState` field after any ACKing sequence, success or fail, RX or TX. Calling `RAIL_AutoAckDisable()` would set all state transitions (RX/TX, success/fail) to idle.	3. Upon entering auto-ACK, the default state (that is, the state to return to after all ACK sequences, success or fail) can now be set separately for TX and RX. That default state is the “success” transition for each of `rxTransitions` and `txTransitions` fields of `RAIL_AutoAckConfig_t`. Upon exiting auto-ACK (that is, when enable = false), transitions are set to `txTransitions` and `rxTransitions`.
4. While in auto-ACK mode, `RAIL_SetTxTransitions()`, `RAIL_SetRxTransitions()`, and `RAIL_SetStateTiming()` could not be called.	4. While auto-ACK is enabled, `RAIL_SetTxTransitions()` and `RAIL_SetRxTransitions()` should still not be called. It is now safe to call `RAIL_SetStateTiming()` while in auto-ACK mode.

8.1 Example Code: RAIL 1.6

The following code outlines how auto-ACKing was enabled in RAIL 1.6. The main interface included `RAIL_AutoAckConfig()` which configured and enabled autoACK and `RAIL_AutoAckDisable()`, which disabled it.

```
RAIL_AutoAckConfig_t autoAckConfig = {
  RAIL_RF_STATE_RX, // Default state after ACKing is RX
  100,              // idleToRx/Tx Timing
  192,              // txToRx/rxToTx timing
  1000              // ACK timeout
};

RAIL_Status_t status = RAIL_ConfigAutoAck(&autoAckConfig);

// If status is RAIL_STATUS_NO_ERROR, then RAIL_AutoAckConfig not only configured
// auto-ACK, but also enabled it.
```

The following code outlines how auto-ACKing was disabled in RAIL 1.6.

```
RAIL_AutoAckDisable();

// Re-configure state transitions that were defaulted back to idle after disabling
// auto-ACK
RAIL_SetRxStateTransitions(RAIL_RF_STATE_RX, RAIL_RF_STATE_RX);
RAIL_SetTxStateTransitions(RAIL_RF_STATE_RX, RAIL_RF_STATE_RX);

// A call to RAIL_SetStateTiming may also be necessary to set timings, as
// they will be left as whatever timings were passed to RAIL_AutoAckConfig.
```

8.2 Example Code: RAIL 2.X

The following code outlines how to enable auto-ACKing in RAIL 2.X. The main interface is now more symmetric, having only `RAIL_ConfigAutoAck()`, which is more symmetric in behavior. The new interface also allows for different state transitions in auto-ACK depending on whether the node is transmitting or receiving the ACK.

```
RAIL_AutoAckConfig_t autoAckConfig = {
  true, /* enable acking */
  1000, /* ack timeout */
  // After receiving an ACK, go to RX, error
  // transition is ignored during auto-ACK
  { RAIL_RF_STATE_RX, RAIL_RF_STATE_RX },
  // After transmitting an ACK, go to RX, error
  // transition is ignored during auto-ACK
  { RAIL_RF_STATE_RX, RAIL_RF_STATE_RX }
};

// State Timings are now handled outside auto-ACK configuration, allowing for more
// granularity in the values set.
RAIL_StateTimig_t stateTiming = { 100, 192 – 10, 100, 192, 0, 0};
RAIL_SetStateTiming(railHandle, &stateTiming);

RAIL_ConfigAutoAck(railHandle, &autoAckConfig);
```

The following code outlines how to disable auto-ACKing in RAIL 2.X.

```
// autoAckConfig is the same autoAckConfig initialized in the example above
autoAckConfig.enable = false;
RAIL_ConfigAutoAck(railHandle, &autoAckConfig);

// State transitions are set to the values specified above during the disable process.
```

9. Event Coalescing

The following table summarizes the event coalescing differences between RAIL 1.6 and RAIL 2.1.

Table 9.1. Event Coalescing Differences between RAIL 1.6 and RAIL 2.1

RAIL 1.6	RAIL 2.1
1. RAIL required an application to implement **all** the `RAILCb_` callbacks, even if the application did not care about or enable the corresponding event(s) for that callback.	1. Applications no longer have to implement any `RAILCb_` callbacks; RAIL will now deal reasonably with an event that has no callback. The set of callbacks is also greatly reduced, and with the exception of `RAILCb_AssertFailed()`, they are registered with RAIL in the relevant API call as follows: • The `RAIL_Init()` `RAIL_InitCompleteionCallbackPtr_t` argument replaces the former `RAILCb_RfReady()`. It can be NULL for no callback. • The `RAIL_SetTimer()` `RAIL_TimerCallback_t` argument replaces the `RAILCb_TimerExpired()` API in RAIL 1.6. It can be NULL for no callback. • The `RAIL_StartRfSense()` `RAIL_RfSense_callbackPtr_t` argument replaces the `RAILCb_RxRadioStatus()` API callback's `RAIL_RX_CONFIG_RF_SENSED` status in RAIL 1.6. It can be NULL for no callback. • The `RAIL_Init()` `RAIL_Config_t` argument's eventsCallback field replaces most other `RAILCb_` callbacks related to radio events in RAIL 1.6. It can be NULL for no callback (though only a very limited application would want to do this). This callback is called whenever any of the events configured via `RAIL_ConfigEvents()` (see item #3) occurs in RAIL. Also see Table 9.2 RAIL 1.6 Callbacks→RAIL 2.1 Callback Equivalents on page 39 for callback and event mapping between RAIL 1.6 and RAIL 2.1. • The `RAIL_ConfigChannels()` `RAIL_RadioConfigChangedCallback_t` argument is a new callback called whenever a radio configuration change occurs. It can be NULL for no callback.
2. When, due to interrupt latency, RAIL saw multiple events at the same time, it would call each individual callback separately in some arbitrary order that may not match the temporal order in which the events actually occurred, or the order the application expected them to be presented. For example, a transmit completion followed by a receive completion, or a transmit LBT success followed by that transmit's completion, could have their respective callbacks called in the opposite order.	2. When latency causes multiple events to be seen simultaneously, the single `RAIL_Config_t.eventsCallback()` callback is called once and presents all the events that have occurred since the last callback, leaving it to the application to use its knowledge of the protocol to sort out the most relevant order in which to process the events and resolve any likely temporal ambiguities.
3. Configuring which RAIL events trigger callbacks was split across several APIs, including `RAIL_RxConfig()`, `RAIL_TxConfig()`, `RAIL_DebugCbConfig()`, and some callbacks could not be disabled. Furthermore, if the application wanted to enable/disable an individual callback, it had to remember the state of the other callbacks in order not to disturb them.	3. `RAIL_ConfigEvents()` is now the central place to control which events will trigger a callback. In addition to the now-obligatory `RAIL_Handle_t` argument, `RAIL_ConfigEvents()` takes two `RAIL_Events_t` arguments: a mask, indicating which events should be changed and the enable/disable values for those events. This allows events to be easily toggled in a thread-safe manner and allows the application to only be called back when a relevant event occurs, instead of every event resulting in a callback.
4. Each `RAILCb_` callback took arguments specific to that callback.	4. The `RAIL_Config_t.eventsCallback()` callback takes a mask of events that occurred. Since it would be quite cumbersome to append additional information for each of those events, RAIL 2.1 takes a different approach. Some of the additional information that used to be passed to the old `RAILCb_` callbacks is now mapped into separate events (such as for the former `RAILCb_RxRadioStatus()` and `RAILCb_TxRadioStatus()` status values). In other cases, some new APIs are provided for the application to retrieve that information, for example, `RAIL_GetRxPacketInfo()`, `RAIL_GetRxPacketDetails()`, `RAIL_GetTxPacketDetails()`, and `RAIL_IEEE802154_GetAddress()`.

9.1 Example Code: RAIL 1.6

A RAIL 1.6 application's event handling callback code might have looked like this:

```
void RAILCb_RfReady(void)
{
  // ... (unused)
}

void *RAILCb_AllocateMemory(uint32_t size)
{
  // ... for Rx packet receive
}

void RAILCb_FreeMemory(void *handle)
{
  // ... for Rx packet receive
}

void *RAILCb_BeginWriteMemory(void *handle,
                              uint32_t offset,
                              uint32_t *available)
{
  // ... for Rx packet receive
}

void RAILCb_EndWriteMemory(void *handle, uint32_t offset, uint32_t size)
{
  // ... for Rx packet receive
}

void RAILCb_RxFifoAlmostFull(uint16_t bytesAvailable)
{
  // ... for Rx packet receive
}

void RAILCb_TxFifoAlmostEmpty(uint16_t spaceAvailable)
{
  // ... (unused)
}

void RAIL_TimerCancel(void)
{
  // ... (unused)
}

void RAILCb_TimerExpired(void)
{
  // ... (unused)
}

void RAILCb_TxPacketSent(RAIL_TxPacketInfo_t *txPacketInfo)
{
  // ... process Tx packet sent
}

void RAILCb_TxRadioStatus(uint8_t status)
{
  // ... process Tx error
}

void RAILCb_RssiAverageDone(int16_t avgRssi)
{
  // ... (unused)
}

void RAILCb_RxPacketReceived(void *rxPacketHandle)
{
  // ... process Rx packet received
}

void RAILCb_RxRadioStatusExt(uint32_t status)
```

```
{
  // ... process Rx error
}

void RAILCb_RxAckTimeout(void)
{
  // ... (unused)
}

void RAILCb_CalNeeded(void)
{
  // ... process calibration
}

void RAILCb_RadioStateChanged(uint8_t state)
{
  // ... (unused)
}

void RAILCb_AssertFailed(uint32_t errorCode)
{
  // ... process assert
}

static void radioInitialize(void)
{
  // ... RAIL_RfInit() and friends

  // Configure RAIL callbacks with receive appended info enabled
  RAIL_RxConfig(RAIL_RX_CONFIG_FRAME_ERROR
                | RAIL_RX_CONFIG_ADDRESS_FILTERED
                | RAIL_RX_CONFIG_BUFFER_OVERFLOW
                | RAIL_RX_CONFIG_PACKET_ABORTED,
                true);
  RAIL_TxConfig(RAIL_TX_CONFIG_BUFFER_UNDERFLOW
                | RAIL_TX_CONFIG_TX_ABORTED);

  // ... other initialization
}
```

9.2 Example Code: RAIL 2.1

A RAIL 2.1 application's event handling code example would now look something like this:

```
// This callback is optional -- the default one will hang
void RAILCb_AssertFailed(uint32_t errorCode)
{
  // ... process assert
}

// A radio event handler is definitely desirable
static void radioEventHandler(RAIL_Handle_t railHandle,
                              RAIL_Events_t events)
{
  // Process multiple events in the most likely order the protocol expects
  // RX events
  if (events & RAIL_EVENT_RX_PACKET_RECEIVED) {
    // ... process Rx packet received
  }
  if (events & (RAIL_EVENT_RX_FRAME_ERROR
                | RAIL_EVENT_RX_FIFO_OVERFLOW
                | RAIL_EVENT_RX_ADDRESS_FILTERED
                | RAIL_EVENT_RX_PACKET_ABORTED)) {
    // ... process Rx error
  }
  // TX events
  if (events & RAIL_EVENT_TX_PACKET_SENT) {
    // ... process Tx packet sent
  }
  if (events & (RAIL_EVENT_TX_UNDERFLOW
                | RAIL_EVENT_TX_ABORTED)) {
    // ... process Tx error
  }
  // Other events
  if (events & RAIL_EVENT_CAL_NEEDED) {
    // ... process calibration
  }
}

static RAIL_Handle_t railHandle = NULL;

static RAIL_Config_t railConfig = { // Must never be const
  .eventsCallback = &radioEventHandler,
  // ...
};

static void radioInitialize(void)
{
  railHandle = RAIL_Init(&railConfig, NULL);

  // ... other initialization

  // Only need to enable and handle events truly of interest
  RAIL_ConfigEvents(railHandle, RAIL_EVENTS_ALL, 0
                    | RAIL_EVENT_RX_PACKET_RECEIVED
                    | RAIL_EVENT_RX_FRAME_ERROR
                    | RAIL_EVENT_RX_FIFO_OVERFLOW
                    | RAIL_EVENT_RX_ADDRESS_FILTERED
                    | RAIL_EVENT_RX_PACKET_ABORTED
                    | RAIL_EVENT_TX_PACKET_SENT
                    | RAIL_EVENT_TX_UNDERFLOW
                    | RAIL_EVENT_TX_ABORTED
                    | RAIL_EVENT_CAL_NEEDED
                   );

  // ... other initialization
}
```

9.3 Callbacks

The following table summarizes RAIL 1.6 callbacks and their RAIL 2.1 callback equivalents.

Table 9.2. RAIL 1.6 Callbacks→RAIL 2.1 Callback Equivalents

RAIL 1.6 Callback	RAIL 2.1 Callback Equivalent
`RAILCb_RfReady()`	Optionally passed into `RAIL_Init()` as its `RAIL_InitCompleteCallbackPtr_t cb` parameter.
`RAILCb_AllocateMemory() RAILCb_FreeMemory() RAILCb_BeginWriteMemory() RAILCb_EndWriteMemory()`	Memory callbacks are no longer used; see `RAILCb_RxPacketReceived()` information below.
`RAILCb_RxFifoAlmostFull()`	`RAIL_Config_t.eventsCallback(, RAIL_EVENT_RX_FIFO_ALMOST_FULL)`; use `RAIL_GetRxPacketInfo()` or `RAIL_GetRxFifoBytesAvailable()`.
`RAILCb_TxFifoAlmostEmpty()`	`RAIL_Config_t.eventsCallback(, RAIL_EVENT_TX_FIFO_ALMOST_EMPTY)`; use `RAIL_GetTxFifoSpaceAvailable()`.
`RAILCb_TimerExpired()`	Passed into `RAIL_SetTimer()` as its `RAIL_TimerCallback_t cb` parameter.
`RAILCb_TxPacketSent()`	`RAIL_Config_t.eventsCallback(, RAIL_EVENT_TX_PACKET_SENT)`; use `RAIL_GetTxPacketDetails()` with `RAIL_TxPacketDetails_t.isAck false / RAIL_CallbackConfig_t.generic(, RAIL_EVENT_TXACK_PACKET_SENT)`; use `RAIL_GetTxPacketDetails()` with `RAIL_TxPacketDetails_t.isAck true`.
`RAILCb_TxRadioStatus(status)`; see status values below.	`RAIL_Config_t.eventsCallback(, event)`; see corresponding events below.
`RAIL_TX_CONFIG_BUFFER_OVERFLOW`	Event removed in RAIL 2.1. APIs prevent Tx overflow from occurring.
`RAIL_TX_CONFIG_BUFFER_UNDERFLOW`	`RAIL_EVENT_TX_UNDERFLOW` and `RAIL_EVENT_TXACK_UNDERFLOW`
`RAIL_TX_CONFIG_CHANNEL_BUSY`	`RAIL_EVENT_TX_CHANNEL_BUSY`
`RAIL_TX_CONFIG_TX_ABORTED`	`RAIL_EVENT_TX_ABORTED` and `RAIL_EVENT_TXACK_ABORTED`
`RAIL_TX_CONFIG_TX_BLOCKED`	`RAIL_EVENT_TX_BLOCKED` and `RAIL_EVENT_TXACK_BLOCKED`
`RAIL_TX_CONFIG_CHANNEL_CLEAR`	`RAIL_EVENT_TX_CHANNEL_CLEAR`
`RAIL_TX_CONFIG_CCA_RETRY`	`RAIL_EVENT_TX_CCA_RETRY`
`RAIL_TX_CONFIG_START_CCA`	`RAIL_EVENT_TX_START_CCA`
`RAILCb_RssiAverageDone()`	`RAIL_Config_t.eventsCallback(, RAIL_EVENT_RSSI_AVERAGE_DONE)`; use `RAIL_GetAverageRssi()`.
`RAILCb_RxPacketReceived()`	`RAIL_Config_t.eventsCallback(, RAIL_EVENT_RX_PACKET_RECEIVED)`; use `RAIL_GetRxPacketInfo(), RAIL_GetRxPacketDetails(), RAIL_PeekRxPacket()` and optionally `RAIL_HoldRxPacket(), RAIL_ReleaseRxPacket(), RAIL_ReadRxFifo()`.
`RAILCb_RxRadioStatus(status)`; see status values below.	`RAIL_Config_t.eventsCallback(, event)`; see corresponding events below.
`RAIL_RX_CONFIG_BUFFER_UNDERFLOW`	Event removed in RAIL 2.1. APIs prevent Rx underflow from occurring.
`RAIL_RX_CONFIG_PREAMBLE_DETECT`	`RAIL_EVENT_RX_PREAMBLE_DETECT`
`RAIL_RX_CONFIG_SYNC1_DETECT`	`RAIL_EVENT_RX_SYNC1_DETECT`
`RAIL_RX_CONFIG_SYNC2_DETECT`	`RAIL_EVENT_RX_SYNC2_DETECT`
`RAIL_RX_CONFIG_FRAME_ERROR`	`RAIL_EVENT_RX_FRAME_ERROR`

RAIL 1.6 Callback	RAIL 2.1 Callback Equivalent
`RAIL_RX_CONFIG_BUFFER_OVERFLOW`	`RAIL_EVENT_RX_FIFO_OVERFLOW`
`RAIL_RX_CONFIG_ADDRESS_FILTERED`	`RAIL_EVENT_RX_ADDRESS_FILTERED`
`RAIL_RX_CONFIG_TIMEOUT`	`RAIL_EVENT_RX_TIMEOUT`
`RAILCb_RxRadioStatusExt(status)`; see status values below.	`RAIL_Config_t.eventsCallback(, event)`; see corresponding events below.
`RAIL_RX_CONFIG_SCHEDULED_RX_END`	`RAIL_EVENT_RX_SCHEDULED_RX_END`
`RAIL_RX_CONFIG_PACKET_ABORTED`	`RAIL_EVENT_RX_PACKET_ABORTED`
`RAIL_RX_CONFIG_FILTER_PASSED`	`RAIL_EVENT_RX_FILTER_PASSED`
`RAIL_RX_CONFIG_TIMING_LOST`	`RAIL_EVENT_RX_TIMING_LOST`
`RAIL_RX_CONFIG_TIMING_DETECT`	`RAIL_EVENT_RX_TIMING_DETECT`
`RAILCb_RxAckTimeout()`	`RAIL_Config_t.eventsCallback(, RAIL_EVENT_RX_ACK_TIMEOUT)`
`RAILCb_CalNeeded()`	`RAIL_Config_t.eventsCallback(, RAIL_EVENT_CAL_NEEDED)`
`RAILCb_AssertFailed(errorCode)`	`RAILCb_AssertFailed(railHandle, errorCode)`
`RAILCb_IEEE802154_DataRequestCommand(data)`	`RAIL_Config_t.eventsCallback(, RAIL_EVENT_RX_DATA_REQUEST_COMMAND); use RAIL_IEEE802154_GetAddress()`
`RAIL_EnableRxFifoThreshold()`	`RAIL_ConfigEvents(railHandle, RAIL_EVENT_RX_FIFO_ALMOST_FULL, RAIL_EVENT_RX_FIFO_ALMOST_FULL)`
`RAIL_DisableRxFifoThreshold()`	`RAIL_ConfigEvents(railHandle, RAIL_EVENT_RX_FIFO_ALMOST_EMPTY, RAIL_EVENTS_NONE)`
`RAIL_TxConfig(cbToEnable)`; see list under `RAILCb_TxRadioStatus()` above.	`RAIL_ConfigEvents(railHandle, mask, events)`; see events under `RAILCb_TxRadioStatus()` above.
`RAIL_RxConfig(cbToEnable, )`; see list under `RAILCb_RxRadioStatus()` above.	`RAIL_ConfigEvents(railHandle, mask, events)`; see events under `RAILCb_RxRadioStatus()` above.
`RAIL_RxConfig(, appendedInfoEnable)`	`RAIL_ConfigRxOptions(, RAIL_RX_OPTION_REMOVE_APPENDED_INFO, (enable) ? 0 : RAIL_RX_OPTION_REMOVE_APPENDED_INFO)`
`RAIL_DebugCbConfig(cbToEnable) RAIL_DEBUG_CONFIG_STATE_CHANGE`	Removed in RAIL 2.1.

10. Packet Receive APIs

The following table summarizes the packet receive API differences between RAIL 1.6 and RAIL 2.x.

Table 10.1. Packet Receive API Differences between RAIL 1.6 and RAIL 2.x

RAIL 1.6	RAIL 2.x
1. Before RAIL 1.6 issued the `RAILCb_RxPacketReceived()` callback to the application, it had to: 1. Copy the entire packet out to user-provided memory through the flexible but somewhat convoluted memory management callback APIs: `RAILCb_AllocateMemory()`, `RAILCb_BeginWriteMemory()`, `RAILCb_EndWriteMemory()`, and `RAILCb_FreeMemory()`. 2. Retrieve appended information, if available, and convert the raw values into RAIL types and timebase (RSSI, timestamp, etc.). 3. All of these tasks took considerable time, entirely in interrupt context.	1. Only a minimum of processing is now performed prior to notifying the application of a receive packet completion, with the application now needing to call `RAIL_GetRxPacketInfo()` and optionally the `RAIL_PeekRxPacket()` and `RAIL_GetRxPacketDetails()` APIs. 1. No packet data is copied. The application can call `RAIL_GetRxPacketInfo()` to quickly obtain the status of the packet and pointers to its data in internal RAIL buffers for direct access or to copy into whatever memory organization the application prefers, at a time best suited to the application. RAIL no longer relies on any memory management callbacks. 2. No details are retrieved or converted until the application calls `RAIL_GetRxPacketDetails()` requesting them. 3. RAIL's interrupt context time is minimized, and while this burden has shifted onto the application, it can now choose to defer processing received packets outside of interrupt context by using the new `RAIL_HoldRxPacket()` API or quickly discard the packet by neither holding it nor copying it to application memory, or explicitly releasing it via the new `RAIL_ReleaseRxPacket()` API.
2. If the application temporarily had no memory to copy out the packet data, that packet data was permanently lost.	2. Packet data can be retained outside of interrupt context via `RAIL_HoldRxPacket()` until explicitly released by the application through `RAIL_ReleaseRxPacket()`. Until a packet is released, its information, data, and details remain available to access via `RAIL_GetRxPacketInfo()`, `RAIL_PeekRxPacket()`, and `RAIL_GetRxPacketDetails()`.
3. Multiple received packets had to be processed in the order received.	3. While the application will still be called back in the temporal order packets are received, it can, through the hold and release mechanism mentioned in item #1.c, choose to process multiple packets in whatever order the application desires. The only consequence of out-of-order processing is reducing the amount of internal space available to receive new packets until the oldest received packet is released.
4. If the internal receive FIFO overflowed, `RAILCb_RxRadioStatus()` would be called on each packet in the FIFO leading up to the one that overflowed	4. The `RAIL_EVENT_RX_FIFO_OVERFLOW` event is now triggered only after the application has been notified of all prior received packet completions.
5. FIFO mode receive was difficult to manage, with the application needing to carefully consume remaining packet data of both successfully and unsuccessfully received packets via `RAIL_ReadRxFifo()`, as well as deal with consuming the appended information of successfully received packets (which included those with CRC errors – see item #6) via `RAIL_ReadRxAppendedInfo()`. In some situations, such an application would need to rely on `RAIL_ResetRxFifo()` to clean up.	5. A FIFO mode application no longer has to worry about how much receive data needs to be consumed from RAIL for unsuccessfully-received packets. RAIL 2.x will automatically flush any remaining data that the application has not already consumed via `RAIL_ReadRxFifo()`. The same goes for appended information on successfully-received packets—it is automatically freed when the packet is released so `RAIL_ReadRxAppendedInfo()` in RAIL 1.6 no longer exists in RAIL 2.x. The `RAIL_PeekRxPacket()` and `RAIL_ReadRxFifo()` APIs no longer permit access to or consumption of appended information data; they will only return true packet payload data. `RAIL_PeekRxPacket()` also now takes a `RAIL_PacketHandle_t` argument allowing the application easy access to newer or incoming packet data beyond the oldest one in RAIL's internal FIFO buffer. While the `RAIL_ResetFifo()` API remains, an application should no longer need to call it and can generally rely on RAIL or `RAIL_ReleaseRxPacket()` to properly manage the application's internal RX FIFO buffer(s).
6. FIFO mode implicitly forced enabling the `RAIL_RX_OPTION_IGNORE_CRC_ERRORS` option (and forcibly disabled it if returning to Packet mode).	6. `RAIL_ConfigData()` no longer implicitly enables `RAIL_RX_OPTION_IGNORE_CRC_ERRORS` for receive FIFO mode, nor implicitly disables it for receive Packet mode.

10.1 Example Code: RAIL 1.6

RAIL 1.6 example code to handle a received packet looked like this:

```
static uint8_t buffer[MAX_PACKET_SIZE + sizeof(RAIL_RxPacketInfo_t)];
static bool isAllocated = false;

void *RAILCb_AllocateMemory(uint32_t size)
{
  int i = 0;
  void *ptr = NULL;
  CORE_DECLARE_IRQ_STATE;

  // We can't support sizes greater than the maximum buffer size
  if (size > (MAX_PACKET_SIZE + sizeof(RAIL_RxPacketInfo_t))) {
    return NULL;
  }

  // Disable interrupts and attempt to grab the buffer
  CORE_ENTER_CRITICAL();
  if (!isAllocated) {
    isAllocated = true;
    ptr = buffer;
  }
  CORE_EXIT_CRITICAL();

  return ptr;
}

void RAILCb_FreeMemory(void *ptr)
{
  CORE_CRITICAL_SECTION(
    isAllocated = false;
  );
}

void *RAILCb_BeginWriteMemory(void *handle, uint32_t offset, uint32_t *available)
{
  if (handle == NULL) {
    return NULL;
  }
  return ((uint8_t*)handle) + offset;
}

void RAILCb_EndWriteMemory(void *handle, uint32_t offset, uint32_t size)
{
  // Do nothing
}

void RAILCb_RxPacketReceived(void *rxPacketHandle)
{
  // rxPacketHandle refers to the place the application
  // provided (via RAILCb_AllocateMemory()) for storing
  // the RAIL_RxPacketInfo_t structure of packet data,
  // info, and details.  That structure contained fields:
  //   RAIL_AppendedInfo_t appendedInfo with fields:
  //     uint32  timeUs                // timestamp at sync detect
  //     bool    crcStatus : 1         // true if passed
  //     bool    frameCodingStatus : 1 // (always true, was never used)
  //     bool    isAck : 1             // true for received ACK
  //     uint8_t subPhy : 1            // usually 0 except for BLE
  //     int8_t  rssiLatch             // packet RSSI
  //     uint8_t lqi                   // packet link quality
  //     uint8_t symcWordId            // sync word 0 or 1
  //  uint16_t dataLength              // number of bytes in dataPtr[]
  //  uint8_t  dataPtr[]               // packet data
}
```

10.2 Example Code: RAIL 2.x

RAIL 2.x example code to handle a received packet looks like this:

```
void appGenericEventHandler(RAIL_Handle_t railHandle, RAIL_Events_t events)
{

  // ... handle some other events

  if (events & RAIL_EVENT_RX_PACKET_RECEIVED) {
    bool getPacketDetails = true;
    RAIL_RxPacketInfo_t packetInfo;
    RAIL_RxPacketHandle_t packetHandle
      = RAIL_GetRxPacketInfo(railHandle, RAIL_RX_PACKET_HANDLE_NEWEST,
                             &packetInfo);
    if (packetHandle != RAIL_RX_PACKET_HANDLE_INVALID) {
      // packetInfo contains basic info about the packet and where
      // in RAIL memory the packet data exists.  Fields are:
      //   RAIL_PacketStatus_t packetStatus which for this event
      //     should be RAIL_RX_PACKET_READY_SUCCESS or
      //     RAIL_RX_PACKET_READY_CRC_ERROR.  Can check this in
      //     lieu of obtaining packet details for its crcPassed.
      //   uint16_t packetBytes corresponds to 1.6's dataLength
      //   uint16_t firstPortionBytes and
      //   uint8_t *firstPortionData points to the first portion of
      //     the packet in RAIL's internal RX FIFO buffer
      //   uint8_t *lastPortionData points to the last portion of
      //     the packet (if any) whose length would be
      //     packetBytes - firstPortionBytes.
      // Can use those pointers to copy or access packet data, e.g.
      //   uint8_t *pktCopy = (uint8_t *) malloc(packetInfo.packetBytes);
      //   if (pktCopy != NULL) {
      //     memcpy(pktCopy, packetInfo.firstPortionData,
      //            packetInfo.firstPortionBytes);
      //     memcpy(pktCopy + packetInfo.firstPortionBytes,
      //            packetInfo.lastPortionData,
      //            packetInfo.packetBytes - packetInfo.firstPortionBytes);
      //   }
      // or can use RAIL_PeekRxPacketBytes(), e.g.
      //   uint8_t *pktCopy = (uint8_t *) malloc(packetInfo.packetBytes);
      //   if (pktCopy != NULL) {
      //     RAIL_PeekRxPacket(railHandle, packetHandle, pktCopy,
      //                       packetInfo.packetBytes, 0);
      //   }

      // Maybe something in the above might affect the decision
      // to getPacketDetails...

      if (getPacketDetails) {
        RAIL_RxPacketDetails_t packetDetails;
        // It's important to fill in where you want the packet timestamp
        // to be calculated within the packet:
        packetDetails.timeReceived.timePosition = RAIL_PACKET_TIME_DEFAULT;
        packetDetails.timeReceived.totalPacketBytes = packetInfo.packetBytes
                        + NUMBER_OF_ON_AIR_BYTES_NOT_INCLUDED_IN_packetBytes;
        if (RAIL_GetRxPacketDetails(railHandle, packetHandle, &packetDetails)
            == RAIL_STATUS_NO_ERROR) {
          // packetDetails contains the packet details.  Fields are:
          //   RAIL_PacketTimeStamp_t timeReceived with fields:
          //     uint32_t packetTime timestamp at the requested timePosition
          //     uint32_t totalPacketBytes as passed in
          //     RAIL_PacketTimePosition_t timePosition corresponding to
          //       the packetTime, based on the position requested
          //   bool    crcPassed  is same as 1.6's crcOk      field
          //   bool    isAck      is same as 1.6's isAck      field
          //   int8_t  rssi       is same as 1.6's rssiLatch  field
          //   uint8_t lqi        is same as 1.6's lqi        field
          //   uint8_t syncWordId is same as 1.6's syncWordId field
          //   uint8_t subPhyId   is same as 1.6's subPhy     field
        }
      }
```

```
      // It's good to release this completed packet when done in order
      // to allow subsequent events handled in this callback to use
      // RAIL_GetRxPacketInfo() with RAIL_RX_PACKET_HANDLE_NEWEST to
      // refer to the next packet in-progress.
      RAIL_ReleaseRxPacket(railHandle, packetHandle);
    }
  }

  // ... handle remaining events

}
```

11. TX and RX APIs

11.1 TX APIs

The following table summarizes the TX API differences between RAIL 1.6 and RAIL 2.x.

Table 11.1. TX API Differences between RAIL 1.6 and RAIL 2.x

RAIL 1.6	RAIL 2.x
1. CCA transmits (CSMA/LBT) and scheduled TX's were done via a `preTxOps` function pointer passed into `RAIL_TxStart()`. Additionally, `RAIL_TxStart()` also took in a pointer to parameters that the function pointer function would take. A normal (that is, non-scheduled, non-CCA) transmit was done by simply passing in a `NULL` function pointer and `NULL` parameters.	1. CCA and scheduled transmits now have their own APIs. `RAIL_StartTx` **always** does normal TX's. `RAIL_StartCcaCsmaTx`, `RAIL_StartCcaLbtTx`, and `RAIL_StartScheduledTx` are no longer passed in as function pointers to another function. They all independently start a TX of the type specified in the name.
2. If you wanted to apply certain options to TX (remove CRC, wait for ack, etc. see `RAIL_TX_OPTION_... #defines`) there was a separate `RAIL_TxStartWithOptions()` that took all the same parameters as `RAIL_TxStart()`, plus a pointer to `RAIL_TxOptions_t` containing the desired options. `RAIL_TxStart()` used default TX options.	2. Now, **all** `RAIL_StartXxxTx()` functions take `RAIL_TxOptions_t` (the value, not a pointer to the value). These options only apply to the transmit they are passed into. Because they all take options, there is no more explicit `RAIL_StartTxWithOptions()`.
3. `RAIL_TxOptions_t` was a structure with explicit fields for each individual option.	3. `RAIL_TxOptions_t` is now a `uint32_t` bitmask. See `RAIL_TX_OPTION_... #defines` in `rail_types.h` for what the bit positions represent.
4. A separate function called `RAIL_SetAbortScheduledTxDuringRx()` decided what to do if a scheduled TX was supposed to happen while another packet was being received. Passing in `true` meant the TX got aborted and `false` meant the TX would be postponed until the RX completed. Whatever behavior the customer set would effect all future scheduled TX's until this function was called again.	4. Behavior regarding what to do if a scheduled TX is supposed to happen while another packet is being received is now indicated by the field `txDuringRx` of `RAIL_ScheduleTxConfig_t` which takes an enum value indicating which of the behaviors should occur. `RAIL_ScheduleTxConfig_t` is then passed into the new `RAIL_StartScheduledTx()` API.
5. To indicate TX completion events, there were two callbacks: `RAILCb_TxPacketSent(void)` and `RAILCb_TxRadioStatus(uint8_t status)`. All successful transmits were reported via a call to `RAILCb_TxPacketSent(void)` and any TX's that failed due to errors were reported to `RAILCb_TxRadioStatus(uint8_t status)` with the `status` variable indicating what type of error occurred. Both ack and non-ack packets were sent to these callbacks and there was no way for the user to know whether or not the callback was due to an ack packet. If there were multiple transmits that occurred before interrupts were handled, these callbacks would still only be called once, once interrupts were handled.	5. All TX events are represented by `RAIL_Event_t` masks passed to the `RAIL_Config_t::eventsCallback()` handler. There are separate events for ack and non-ack packets which indicate whether or not the transmit was an ack, and whether the transmit was successful, aborted, blocked, or failed due to a buffer underflow. Before interrupts are handled, a maximum of one of each transmit type (ack/non-ack) can be queued up to be reported the next time the `RAIL_Config_t::eventsCallback()` handler is called. In other words, the `RAIL_Event_t` mask passed to the `RAIL_Config_t::eventsCallback()` handler will contain up to one of `RAIL_EVENT_TX_PACKET_SENT`, `RAIL_EVENT_TX_PACKET_ABORTED`, `RAIL_EVENT_TX_PACKET_BLOCKED`, `RAIL_EVENT_TX_PACKET_UNDERFLOW` and/or up to one of `RAIL_EVENT_TXACK_PACKET_SENT`, `RAIL_EVENT_TXACK_PACKET_ABORTED`, `RAIL_EVENT_TXACK_PACKET_BLOCKED`, `RAIL_EVENT_TXACK_PACKET_UNDERFLOW`.

11.1.1 Example Code: RAIL 1.6

The following code shows how a CSMA transmit with some options was done in RAIL 1.6.

```
// Set a preTxOperation to be passed into RAIL_StartTx()
RAIL_PreTxOp_t preTxOp = &RAIL_CcaCsma;

// Specify TX options, which were a represented by a structure in 1.6
RAIL_TxOptions_t txOptions = {
  false, // Do not wait for an ACK after transmitting
  true,  // Do not send a CRC with this transmit
  0      // Use Sync Word 0
};

// Specify LBT parameters
RAIL_CsmaConfig_t csmaConfig = {
  0,   // Used for fixed backoff
  0,   // Used for fixed backoff
  1,   // Single try
  -75, // Override if not desired choice
  0,   // No backoff (override with fixed value)
  128, // Override if not desired length
  0,   // no timeout
};

// Start a TX on channel 0 with the configurations specified above
RAIL_TxStartWithOptions(0, &txOptions, preTxOp, &preTxOpParams);
```

11.1.2 Example Code: RAIL 2.x

The function pointers were removed in RAIL 2.x so an LBT transmit is done simply by calling the `RAIL_StartCcaLbtTx()` endpoint and passing in desired options and parameters. Additionally, TX options are now specified as bitmasks instead of structures.

```
RAIL_TxOptions_t txOptions = RAIL_TX_OPTION_REMOVE_CRC;

// Specify CSMA parameters
RAIL_CsmaConfig_t csmaConfig = {
  0,   // Used for fixed backoff
  0,   // Used for fixed backoff
  1,   // Single try
  -75, // Override if not desired choice
  0,   // No backoff (override with fixed value)
  128, // Override if not desired length
  0,   // no timeout
};


// Assuming this is single-protocol RAIL, no need to pass in RAIL_SchedulerInfo_t
RAIL_StartCcaCsmaTx(railHandle, 0, txOptions, &csmaConfig, NULL);
```

11.2 RX APIs

The following table summarizes the RX API differences between RAIL 1.6 and RAIL 2.x.

Table 11.2. RX API Differences between RAIL 1.6 and RAIL 2.x

RAIL 1.6	RAIL 2.x
1. `RAIL_SetRxOptions()` took a 32-bit integer that contained a bitfield of receive options, with only a few options configured.	1. `RAIL_ConfigRxOptions()` takes two `RAIL_RxOptions_t` arguments: a mask indicating which options should be changed, and the options to be set in the changed fields. This means that options can be easily toggled in a thread-safe manner. There are also more options that are configured using this API.
2. `RAIL_RxConfig()` had a boolean argument that determined whether appended information would be included with received packets.	2. `RAIL_RxOptions_t` has a `RAIL_RX_OPTION_REMOVE_APPENDED_INFO` bit. This bit is not set by default, which means that appended information is still included by default.

11.2.1 Example Code: RAIL 1.6

The following code shows how RX options were configured in RAIL 1.6. The disadvantage was that there was no way to affect only one option. That is, all options would be written with something upon every call to `RAIL_SetRxOptions()`.

```
// Set up RX options to store the CRC
// This has the side effect of setting other RX options (RAIL_RX_OPTION_IGNORE_CRC_ERRORS
// and RAIL_RX_OPTION_ENABLE_DUALSYNC) 0/false.
RAIL_SetRxOptions(RAIL_RX_OPTION_STORE_CRC);
```

11.2.2 Example Code: RAIL 2.x

The `RAIL_ConfigRxOptions` in RAIL 2.x allows you to set all **or** just a subset of the RX options. This code example shows how to use this feature.

```
// This function call indicates that only the “store CRC” option will be affected,
// and it will be set to 1/true. All other // options will retain the same values
// they had before this call
RAIL_ConfigRxOptions(railHandle, RAIL_RX_OPTION_STORE_CRC, RAIL_RX_OPTION_STORE_CRC);

// If you just want to mimic the exact behavior of RAIL 1.6 RAIL_SetRxOptions, as in
// the previous example, just use RAIL_RX_OPTIONS_ALL as the mask. This will set
// “store CRC” to true and all other options to 0/false.
RAIL_ConfigRxOptions(railHandle, RAIL_RX_OPTIONS_ALL, RAIL_RX_OPTION_STORE_CRC);
```

12. RSSI

The following table summarizes the Received Signal Strength Indicator (RSSI) differences between RAIL 1.6 and RAIL 2.x.

Table 12.1. RSSI Differences between RAIL 1.6 and RAIL 2.x

RAIL 1.6	RAIL 2.x
1. There were three ways to get RSSI: 1. `RAIL_GetRssi()`: Immediately returned the instantaneous RSSI value for the current receive channel. If receive is off or no signal has been received, this returned `RAIL_RSSI_INVALID`. 2. `RAIL_PollAverageRssi()`: A blocking function that repeatedly read the RSSI register on the current RX channel and returned the average of those readings. Any invalid readings were not included in the average. 3. `RAIL_StartAverageRSSI()`: A non-blocking function that averaged the RSSI on the specified channel for the specified time. `RAILCb_RssiAverageDone()` would be called when the period completed, and then the value could be read via `RAIL_GetAverageRSSI()`. `RAIL_AverageRSSIReady()` would also indicate whether the value was ready to be read via `RAILCb_RssiAverageDone()`. Calling `RAIL_GetAverageRSSI()` before the value was ready returned `RAIL_RSSI_INVALID`.	1. `RAIL_PollAverageRSSI()` was eliminated because few customers used it and it was slightly redundant. If a blocking function is needed, you can simply call `RAIL_StartAverageRssi()` and then repeatedly check `RAIL_IsAverageRssiReady()` in an infinite loop, until is returns `true`. Upon returning `true`, you would then call `RAIL_GetAverageRssi()`. The remaining RSSI-related functions are: 1. `RAIL_GetRssi()`: Returns the RSSI of the current RX channel instantly or `RAIL_RSSI_INVALID` if no value is available. If you are using Silicon Labs Dynamic Multiprotocol, this function will also return `RAIL_RSSI_INVALID` immediately if the provided `RAIL_Handle_t` is not active. Specifying `wait` as `true` will cause the function to block and wait until a valid RSSI is available and return that value (that is, it will never return `RAIL_RSSI_INVALID` if `wait` is `true`). As long as your application is in RX mode when calling this function (which it **always** should be), this function will generally return within a few symbol times. **When using dynamic multiprotocol,** `wait` **must always be** `false`, as indefinitely blocking functions are generally not safe. 2. `RAIL_StartAverageRssi()`, which will trigger the `RAIL_EVENT_RSSI_AVERAGE_DONE` when complete. After that event (or after `RAIL_IsAverageRssiReady()` returns `true`), it is safe to call `RAIL_GetAverageRssi()` and expect a valid RSSI value. Calling `RAIL_GetAverageRssi()` before the `RAIL_EVENT_RSSI_AVERAGE` event will return `RAIL_RSSI_INVALID`.

12.1 Example Code: RAIL 1.6

The only piece of RSSI functionality in RAIL 1.6 that was removed was a specific API for blocking average RSSI. Blocking average RSSI was done like this in RAIL 1.6:

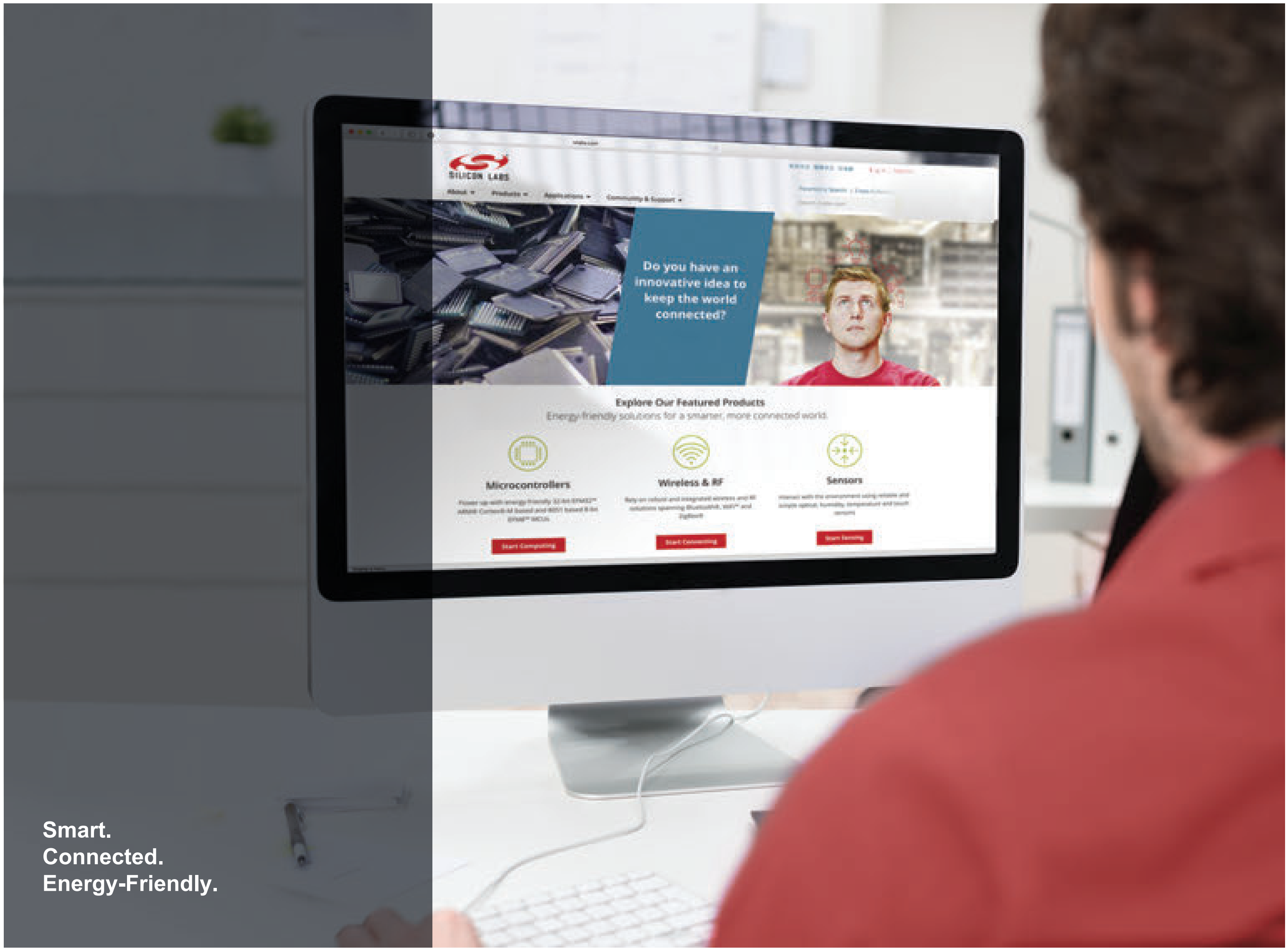
```
// Take an average RSSI value over 10000us
uint16_t rssiResult = RAIL_PollAverageRSSI(10000);
```

12.2 Example Code: RAIL 2.x

The RSSI blocking function was removed in RAIL 2.x to reduce library size. If the RSSI blocking function is needed, implement it like this (single protocol only, never with dynamic multiprotocol):

```
uint16_t rssiResult = RAIL_RSSI_INVALID;

if (RAIL_StartAverageRssi(railHandle, 0, 10000, NULL) == RAIL_STATUS_NO_ERROR) {
  while(!RAIL_IsAverageRssiReady(railHandle)) ;
  rssiResult = RAIL_GetAverageRssi(railHandle);
};
```



Products
www.silabs.com/products

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com