UG305: Dynamic Multiprotocol User's Guide

This version of UG305 has been deprecated.

For the latest version, see docs.silabs.com.

****

This user's guide provides details about implementing an application using Silicon Labs' Dynamic Multiprotocol solution. Dynamic multiprotocol time-slices the radio and rapidly changes configurations to enable different wireless protocols to operate reliably at the same time.

KEY POINTS

- Dynamic Multiprotocol architecture
- About the Radio Scheduler
- Radio Scheduler examples
- Protocol-specific implementation notes

1. Introduction

This document describes how Silicon Labs software is designed to be used by multiple protocols on a single wireless chip. Dynamic multiprotocol time-slices the radio and rapidly changes configurations to enable different wireless protocols to operate reliably at the same time.

Note: The Zigbee-specific information in this document applies to version 6.10.x and lower.

Details on specific dynamic multiprotocol implementations are provided in the following application notes:

AN1133: Dynamic Multiprotocol Development with Bluetooth and Zigbee EmberZNet SDK 6.x and Lower

AN1134: Dynamic Multiprotocol Development with Bluetooth and Proprietary Protocols on RAIL in GSDK v2.x

AN1269: Dynamic Multiprotocol Development with Bluetooth® and Proprietary Protocols on RAIL in GSDK v3.x and Higher

AN1209: Dynamic Multiprotocol Development with Bluetooth and Connect

AN1265: Dynamic Multiprotocol Development with Bluetooth® and OpenThread in GSDK v3.x

1.1 Terminology

The following lists some of the terminology specific to the dynamic multiprotocol implementation.

Radio Abstraction Interface Layer (RAIL): The common API through which higher level code gains access to the EFR32 radio.

Radio Operation: A specific action to be scheduled. A radio operation has both a radio configuration and a priority. Each stack can request that the radio scheduler perform up to two radio operations (background receive and either Scheduled Receive or Scheduled transmit) at a time:

- **Background Receive**: Persistent receive, intended to be interrupted by Scheduled operations, and returned to after their completion.
- **Scheduled Receive:** Receive packets or calculate RSSI at a specified time and duration. (Developers working on RAIL, note that in terms of the RAIL API, "Scheduled Receive" as used in this document refers to any receive operation, other than `RAIL_StartRx`, and is not just limited in scope to `RAIL_ScheduleRx`.)
- **Scheduled Transmit:** Any one of various transmit operations including immediate transmit, scheduled (future) transmit, or CCA-dependent transmit. (Developers working on RAIL, note that in terms of the RAIL API, "Scheduled Transmit" as used in this document refers to any transmit operation, and is not limited in scope to `RAIL_StartScheduledTx`.)

Radio Config: Determines the state of the hardware that must be used to perform a radio operation.

Radio Scheduler: RAIL component that arbitrates between different protocols to determine which will have access to the radio.

Priority: Each operation from each stack has a default priority. An application can change default priorities.

Slip Time: The maximum time in the future when the operation can be started if it cannot begin at the requested start time.

Yield: A stack must voluntarily yield at the end of an operation or sequence of operations, unless it is performing a background receive. Until the stack yields, the scheduler will not scheduler lower priority tasks.

RTOS (Real Time Operating System) Kernel: The part of the operating system that is responsible for task management, and inter-task communication and synchronization.

1.2 Architecture

Dynamic Multiprotocol makes use of the EFR32 hardware and the RAIL software as its building blocks. Zigbee, Bluetooth, and/or any other standards-based or proprietary protocols can then be built on top of these foundational layers, using Micrium to manage execution of code between different protocols. The following diagram illustrates the general structure of the software modules.
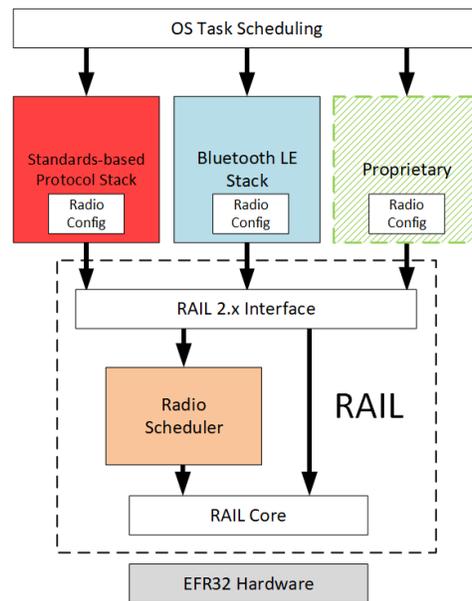

Figure 1.1. General Dynamic Multiprotocol Software Architecture

Beginning with version 2.0, RAIL requires the passing of a radio configuration handle to the RAIL API calls. This configuration describes various PHY parameters that are used by the stack.

The OS is an RTOS that allows stacks and application logic to share CPU execution time.

The Radio scheduler is a software library that intelligently answers requests by the stacks to perform radio operations to maximize reliability and minimize latency. API's provided by RAIL that do not engage the radio bypass the Radio scheduler.

The RAIL core configures the EFR32 hardware in response to instructions from the radio scheduler.

1.3 Single Firmware Image

Dynamic Multiprotocol allows a software developer to generate a single monolithic binary that is loaded onto an EFR32. Software updates are done by upgrading the entire binary. This is accomplished using the Gecko Bootloader, the details of which can be found in *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower* and *UG489: Silicon LabsGecko Bootloader User's Guide for GSDK 4.0 and Higher*.

1.4 Independent Stack Operation

The Silicon Labs stacks still operate independently of one another in a Dynamic Multiprotocol situation. Certain long-lived radio operations will have an impact on another protocol's latency and compliant operation. It is up to the application to determine any special considerations for these events. See section 2. The Radio Scheduler for more information.

2. The Radio Scheduler

The Radio Scheduler is a component of RAIL (Radio Abstraction Interface Layer). RAIL provides an intuitive, easily-customizable radio interface layer and API, which supports proprietary or standards-based wireless protocols. The Radio Scheduler is designed to allow for radio operations that can be scheduled and prioritized. Different radio operations in each protocol may be more or less important, or more or less time sensitive, depending on the situation. The scheduler can take those into account when making decisions about conflicts and how to adjudicate them.

Unless you are developing applications with a custom protocol on RAIL, most radio scheduler functions are handled automatically by underlying stack and RAIL code. You only need to use the stack through its normal API.

At a high level, the stack sends a radio operation (for example a Scheduled Receive or Scheduled Transmit). The radio operations are queued and then serviced at a future time based upon their parameters. When it is time to start the radio operation the scheduler examines whether a competing event exists and whether the operation can be delayed. If the scheduler cannot run the event, it returns the result to the higher layer, which may retry with new parameters.

Once the radio operation has started, the corresponding stack can send the scheduler additional operations based on the results of the previous operation (for example waiting for an ACK). At the end of each operation or sequence of operations the stack must yield use of the radio.

2.1 Radio Operations

Each event in the scheduler is broken up into elements called Radio Operations, which are associated with a radio config and a priority.

Every operation has a priority and is interrupted if the scheduler receives a higher priority operation that overlaps in time. Lower priority radio operations that cannot be run based on their schedule parameters will fail, and it is up to the respective stack to retry them. Once the scheduler actively runs a radio operation from the stack, the stack can continue to send additional radio operations until it voluntarily yields, or until the scheduler receives a higher priority radio operation and preempts it.

- Background Receive
- Scheduled Receive
- Scheduled Transmit

Each stack can ask the Radio Scheduler to perform up to two radio operations (background receive and either Scheduled Receive or Scheduled transmit) at a time:

Each operation has the following parameters:

Parameter	Description
Start Time	An indication at what point in the future this radio operation will run. This could be "run right now" or some value in microseconds in the future.
Priority	A number that indicates the relative priority of the operation. When using the default settings, Bluetooth LE radio operations are almost always higher priority than Zigbee operations.
Slip Time	An amount of time that the event can be delayed beyond its start time and still be acceptable to the stack. This may be 0, in which case the event cannot be slipped.
Transaction Time	The approximate amount of time that it takes to complete the transaction. Transmit events usually have a much more well-defined transaction time, while receive events are often unknown. This is used to help the radio scheduler determine whether an event can be run.

The stack defines these various parameters appropriate to the operation being executed. For example, Bluetooth connection events are often scheduled in the future and have no allowed slip, whereas Zigbee transmit events can often be delayed a small amount and start later.

From the perspective of the RAIL Radio Scheduler, Scheduled transmit and Scheduled receive are identical. They are both simply operations that require use of the radio, and thus cannot be executed simultaneously. The difference is only apparent at RAIL API layer, where either a TX or RX API is called.

2.1.1 Background Receive

This is a continuous receive mode that is intended to be interrupted by other operations and returned to after their completion. If Background Receive is higher priority than other operations, those radio operations will not be scheduled and will not run. It is up to the stacks or application to change the priority or voluntarily yield. See section 5.1 Examples with Background Receive, Yield Radio and State Transition for examples of how Background receive interacts with Scheduled operations.

2.1.2 Scheduled Receive

This is a receive at a future time with a specified duration. The radio scheduler takes into consideration the radio switching time in deciding whether the operation will be scheduled. If it cannot be scheduled, then the scheduler sends a fail event to the calling stack. The radio operation is automatically extended until the stack voluntarily yields, or the scheduler receives a higher priority operation and interrupts it. Extending the receive allows the stack to continue a radio operation based on the requirements of the higher level protocol, for example transmission of a response based on the received data.

2.1.3 Scheduled Transmit

This is a transmit at a future time with a minimum duration. This minimum duration can include expected follow-on events, for example an ACK to an IEEE 802.15.4 transmit. However, the minimum time for this operation does not have to include unexpected events that may extend the time beyond the minimum duration, for example backoffs due to CCA failures in IEEE 802.15.4. The radio scheduler takes into consideration the radio switching time in deciding whether the operation will be scheduled. If it cannot be scheduled, then the scheduler sends a fail event to the calling stack.

2.2 Radio Config

Each radio operation is associated with a predefined radio config that determines the state of the hardware that must be used to perform the operation. The Radio Configs keep track of the stack's current state so that future radio operations will use the same radio parameters. Radio Configs may be active or dormant. If the stack changes an active Radio Config then RAIL makes an immediate change to the hardware configuration as well, for example changing a channel. If the radio config is not currently active then the next scheduled radio operation will use the new radio config.

2.3 Priority

Each radio operation has a priority which indicates to the scheduler which operation should be executed if there is a timing overlap between multiple operations. The scheduler treats a priority of 0 as the highest priority and 255 as the lowest priority. The radio scheduler will allow the task with the highest priority to access the physical radio hardware. With most tasks control is returned to the radio scheduler only on completion, but tasks like background receive will be interrupted in case a task with higher priority becomes active.

The stacks each have a default set of priorities based on Silicon Labs' analysis of how best to cooperate to maximize the duty cycle and avoid dropped connections for a generic use case. Specific use cases may have different needs. The priorities are as follows, from highest to lowest:

1. Bluetooth LE Scheduled Transmit
2. Bluetooth LE Scheduled Receive
3. Other protocol Scheduled Transmit
4. Other protocol Background Receive

These priorities may be overridden or changed by the application. It is up to the application to decide under what circumstances to change them. Section 4.2 802.15.4 RAIL Priority and section 6.1 Bluetooth Priorities contain more details on priorities for their specific instances.

2.4 Slip Time

Every radio operation must have a "slip time", or maximum start time, meaning the furthest time in the future when the operation can be started if it cannot begin at the requested start time. This allows for the scheduler to work around higher priority events that are occurring at the same time, or higher priority events that extend beyond their expected duration. The protocol generally dictates what the slip time can be, but the radio scheduler is capable of handling this on a per-operation basis, allowing a stack to slip some events but not others. In general, IEEE 802.15.4 has longer slip time and Bluetooth LE has a minimal slip time.

2.5 Yield

Once a sequence of radio operations is actively being run, the stack may continue to add operations extending the initial operation until the stack has nothing more to do for the particular message exchange. A stack must voluntarily yield unless it is performing a background receive. If a stack does not yield then it will continue to extend its radio operation, and lower priority radio operations will then trigger a failure back to the corresponding stack that requested that radio operation. A higher priority operation cannot interrupt a currently-running, lower priority radio operation that has not yielded. See section 5.1 Examples with Background Receive, Yield Radio and State Transition for examples of situations where explicitly yielding the radio is necessary.

2.6 Interrupting a Radio Operation

A scheduled radio operation may be interrupted if a higher priority operation conflicts with it. This could occur in the following two circumstances:

1. A scheduled radio operation takes longer than expected and the corresponding stack does not yield before the higher priority radio operation must start.
2. A higher priority radio operation has just been scheduled to occur in the future and conflicts with a lower priority operation already scheduled.

2.7 Long-Lived Radio Operations

Certain long-lived Radio Operations can have an outsized impact on the correct operation of the product. The application may need to coordinate these operations between the protocols. If the application does not, then the radio scheduler priorities take precedence. For example, an IEEE 802.15.4 energy scan can require that the radio stay on to gather sufficient energy readings. If the application does not properly coordinate the operations, the scan could be interrupted prematurely due to a higher priority Bluetooth operation.

3. Radio Scheduler Examples

All examples use Bluetooth LE and Zigbee, but the principles apply to other Bluetooth/802.15.4 combinations.

The scheduler starts out by having a low priority Zigbee background receive operation. This represents an always-on router that may need to receive IEEE 802.15.4 packets at unknown times. A Bluetooth LE connection is also active and requires the stack to be ready to receive every 30 ms. The Bluetooth LE stack may schedule this well in advance due to the connection's predictable nature.

3.1 Priority Scheduling

This provides a basic example of adjudicating priorities of the different radio operations.
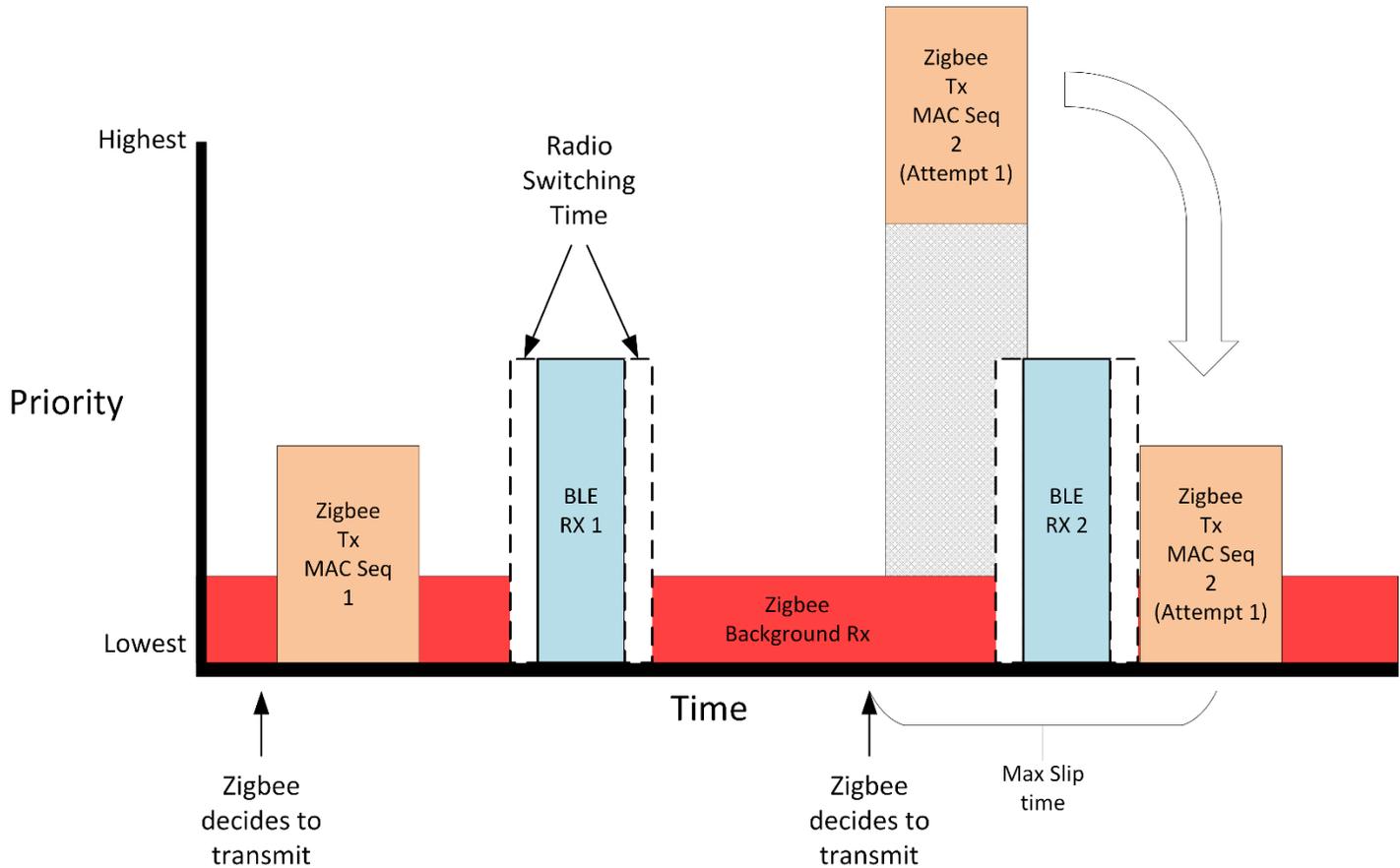
The Zigbee stack decides that it needs to send a packet. It may do this as an on-demand event, meaning the stack decides that it wants to send a packet *now* without informing the scheduler well in advance. This is in contrast to how Bluetooth LE operates, where the scheduled operations are known reasonably far in advance. The scheduler evaluates that it is possible to perform the Zigbee TX 1 radio operation and still service the higher priority Bluetooth LE reception event in the future. Therefore, the scheduler allows the transmit event to occur. The Zigbee stack performs all the pieces of this transmit operation (waiting for a MAC ack) and then voluntarily yields. The estimated transaction time of the Zigbee transmit radio operation does NOT include retries.

In this example, Bluetooth LE is *already* scheduled to receive in the future and the Zigbee stack wants to transmit. For the first Zigbee TX 1 radio operation there is enough time before the Bluetooth LE RX 1 radio operation, so the scheduler allows the stack to perform the operation. Later, when the Zigbee stack tries to schedule Zigbee TX 2 the scheduler determines there is not enough time before the high priority Bluetooth LE RX 2 event. However, the Zigbee stack has indicated that this action may slip its start time. The radio scheduler determines that given the expected duration of the Bluetooth LE radio operation the Zigbee operation can start after that event and still be within the slip time indicated by the Zigbee stack.

If all goes as expected, the Zigbee transmit operation will have its first attempt occur without any failures due to scheduling.

3.2 Priority Interruption Example

This example illustrates a higher priority operation interrupting a lower priority one.
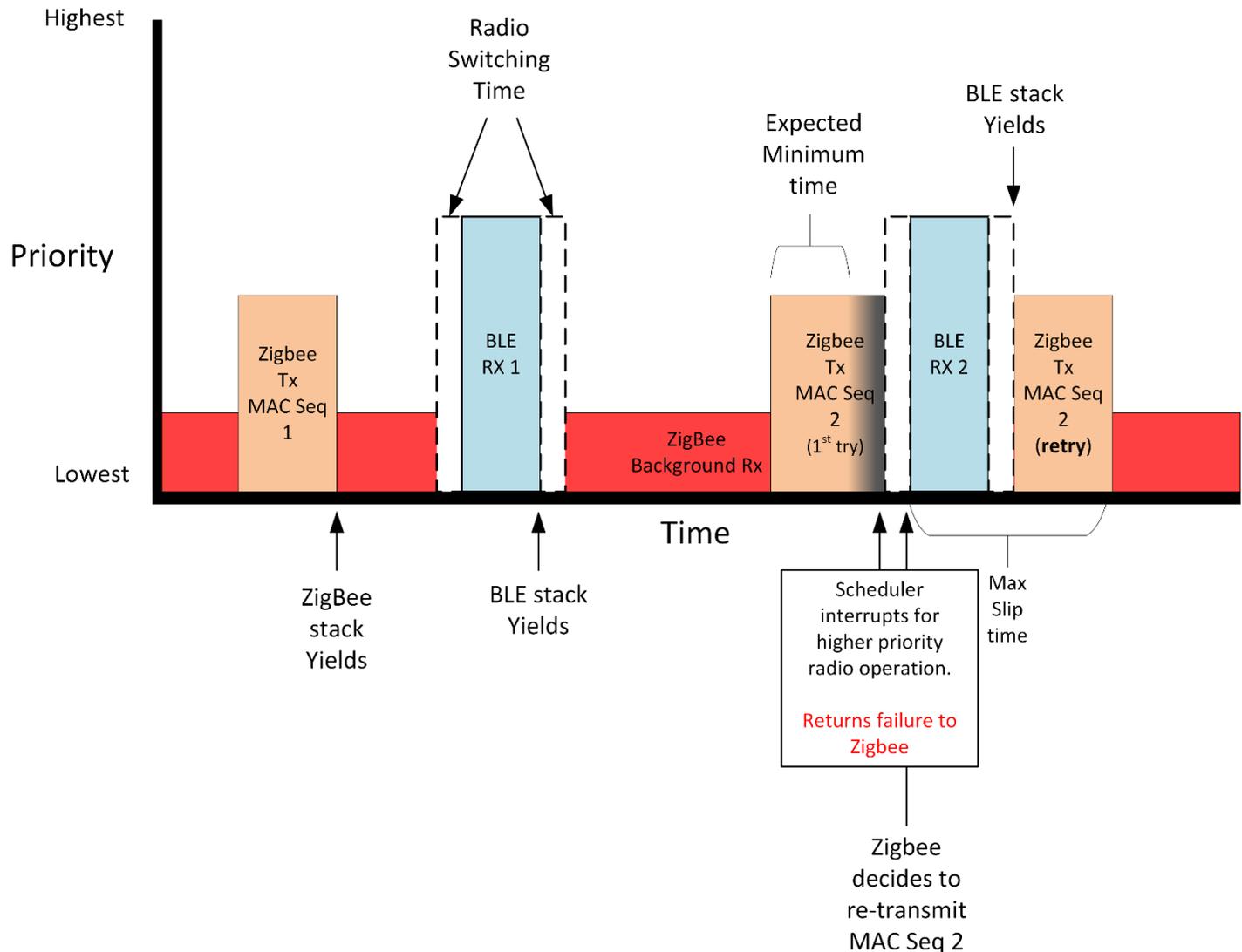

This example starts in the same way as the previous example. Zigbee and Bluetooth LE both have a radio operation that is scheduled without any collision.

Later, the Zigbee stack decides it wants to send another packet for the Zigbee TX 2 event. The scheduler determines that it should be possible to schedule this event and service the Bluetooth LE RX 2 event later, based on the minimum time that the Zigbee TX 2 event must take. However, the Zigbee TX 2 event takes longer than expected due to a long random backoff and does not yield in time. This causes the event to collide with a higher priority radio operation, and so the Radio Scheduler interrupts the Zigbee event and returns a failure to the higher-level stack. The Bluetooth LE event occurs normally and when it is complete it voluntarily yields to any lower-priority operations.

Upon receiving the failure from the radio scheduler, the Zigbee stack immediately attempts to retry the MAC message. It schedules the operation and includes a slip time. At this point the Bluetooth LE stack has priority over the radio and thus the operation cannot be started yet, but the scheduler accepts the new radio operation. The Bluetooth LE stack completes its scheduled receive and yields the radio. The scheduler then triggers the Zigbee transmit operation to occur because it is still within the slip time of the initial start operation. After the transmit completes the scheduler returns to the background receive operation.

3.3 Higher Priority Operation that is Extended

This example shows what happens when a higher priority operation takes longer than originally anticipated and causes a lower priority operation to miss its opportunity.
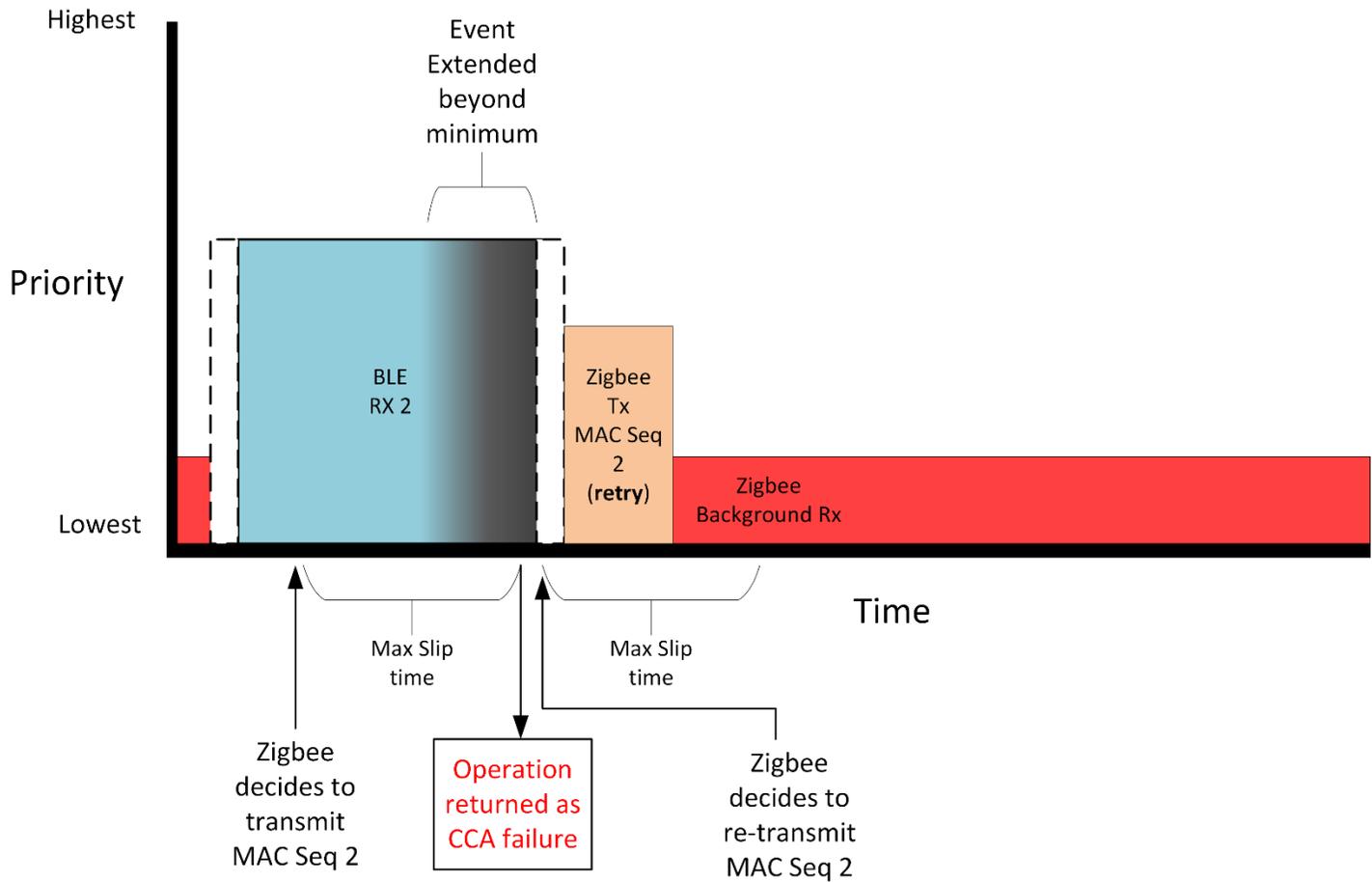
In this case, Bluetooth LE has a Scheduled receive that is currently taking place. Zigbee decides to send a packet but it cannot be run right now. The scheduler accepts the operation under the assumption that the Bluetooth LE event will complete before the end of the slip time of the Zigbee event. However, the Bluetooth LE event extends longer since additional packets are sent between the devices. The Bluetooth LE operation has priority, so the Zigbee operation eventually runs out of slip. An error is returned to the stack. Zigbee decides to re-transmit the packet. Again, the Zigbee stack indicates the operation should start now but may slip into the future. The scheduler is in the middle of changing the radio config so it cannot begin the operation immediately. Instead, it slips the radio operation start time a small amount and then executes the operation.

3.4 Higher Priority Operation Without Interruption

In this example the radio scheduler is running on a node acting as a Bluetooth LE peripheral and that node has a number of connec-tions to different central devices. It also has a periodic advertising beacon that is transmitted. The following figure shows a case where these events are occurring virtually back-to-back and do not allow for enough time to switch back to the Zigbee radio config. Therefore it will create a period where the Zigbee stack is unable to transmit even with the slip time.
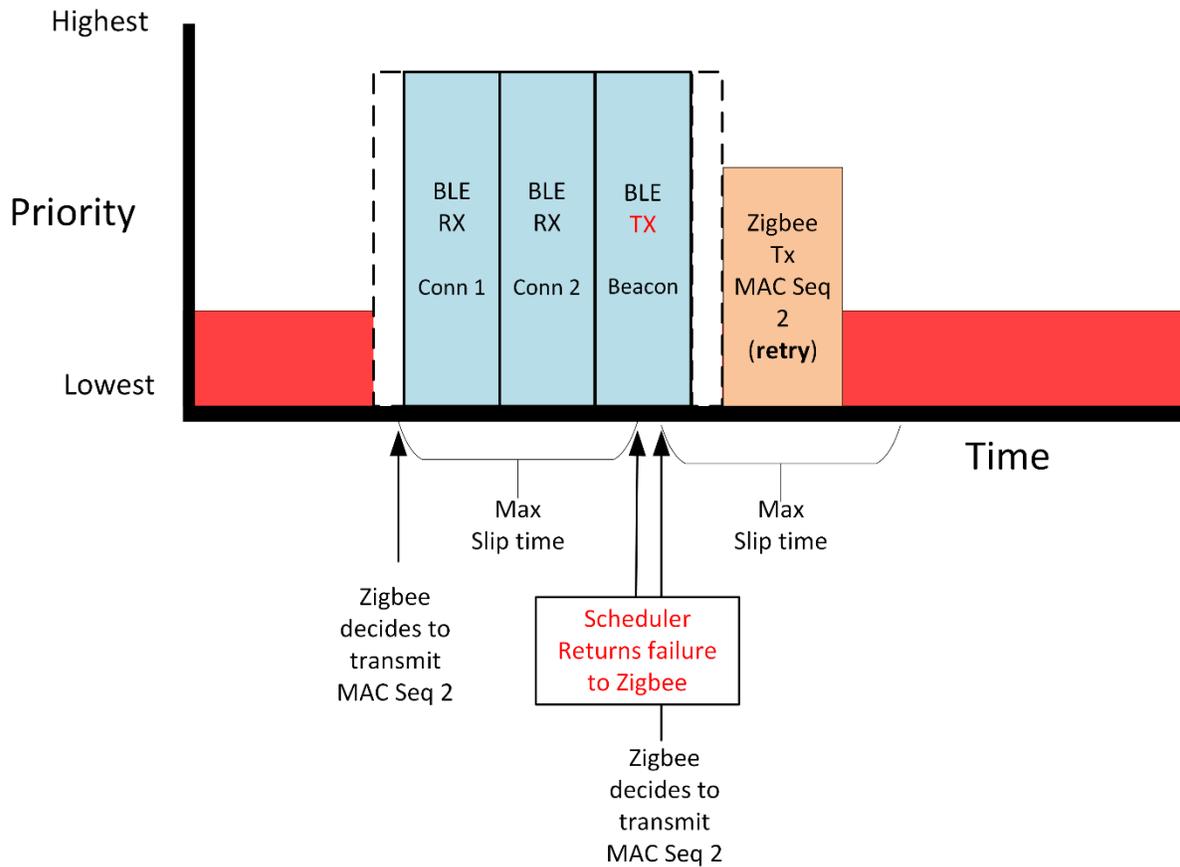
Zigbee asks the scheduler to schedule a transmit radio operation. Even though the scheduler knows that the event will fail due to scheduled higher priority operations, it still accepts the scheduled event. This is done for two reasons. First, circumstances may change and the event can be executed. Second, the stack sitting on top of the radio scheduler may try to retry the action. If the result of the failed scheduling was returned immediately then the stack's attempt to retry would be unlikely to succeed since no time has passed. Instead, by queuing the event and returning the failure *after* the slip time has expired, a retry (with its own slip time) has a better chance of success as the set of upcoming radio operations will be different.

3.5 Receive When a Higher Priority Operation is Running

This example illustrates what happens when Bluetooth LE is active and a lower priority operation will be receiving data.
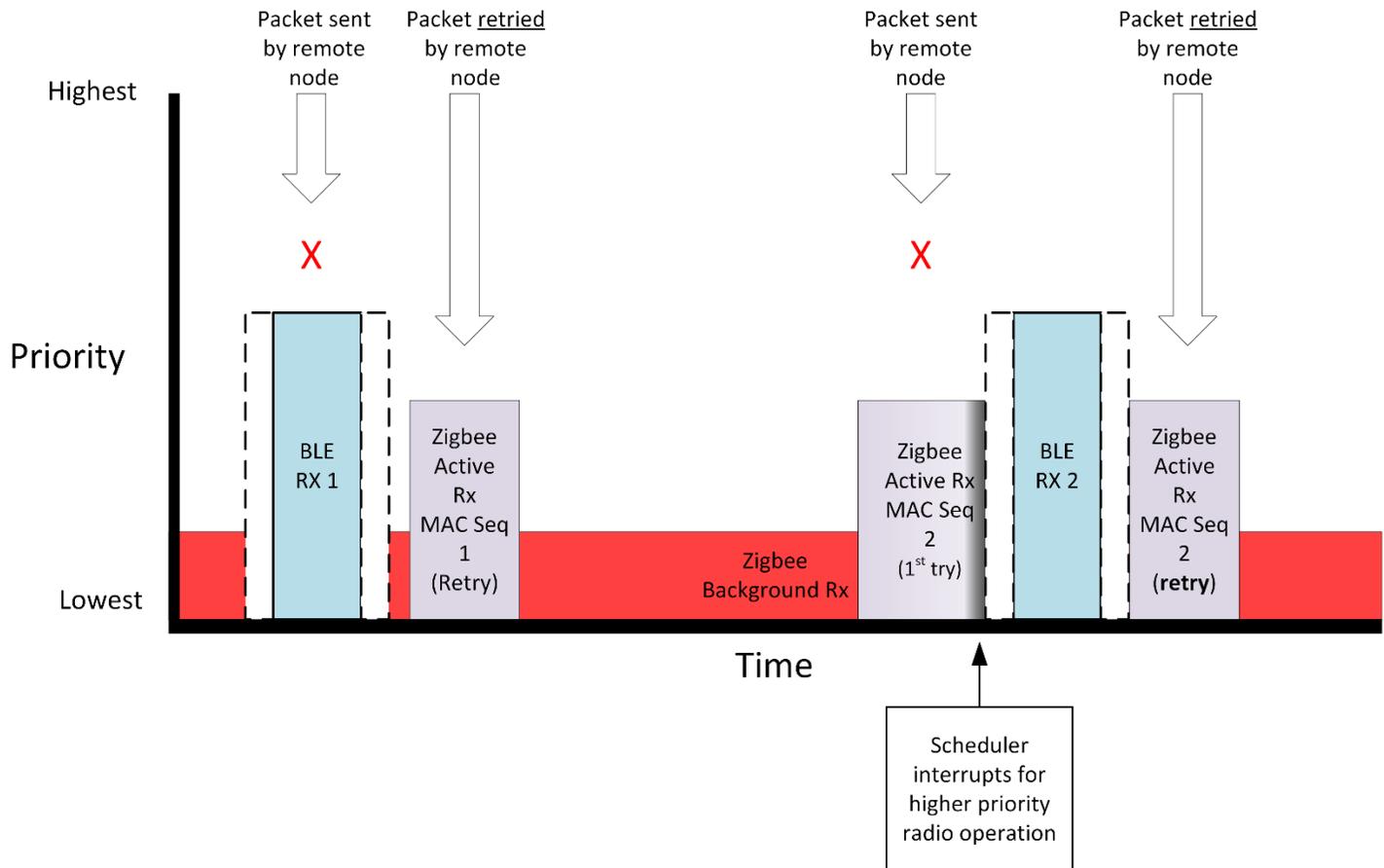
In the first case, when an IEEE 802.15.4 message is sent and the Bluetooth LE stack is utilizing the radio for an active receive the Zigbee stack will not be online to receive the message. However, the Zigbee sender of the message will retry in most cases and with backoffs and other timing alterations is not going to conflict with another higher priority scheduled Bluetooth receive events unlikely to collide. The Zigbee message is received successfully.

The second case shows that, in the case of an active receive, the Zigbee stack may still be interrupted and not receive (or ACK) the message. Successful communication relies on retries at the MAC or higher layer to send this message again and verify the Dynamic Multiprotocol device receives the message.

While there may be considerations for whether or not active receive should be interrupted, it is difficult for the scheduler to make that determination. In general, the robustness of the protocols should allow for messages to be successfully received even with interruptions.

4. Implementing Multiprotocol with an 802.15.4-Based Stack

This chapter offers general information about implementing an 802.15.4-based stack such as Zigbee or Connect as part of a multiprotocol applications. For specifics on how to configure plugins and other details specific to a particular protocol, see one of the following application notes:

- *AN1133: Dynamic Multiprotocol Development with Bluetooth and Zigbee EmberZNet SDK 6.x and Lower*
- *AN1209: Dynamic Multiprotocol Development with Bluetooth and Connect*

4.1 Wireless Protocol Support

Different wireless protocols have different characteristics that have been leveraged with the design of Dynamic Multiprotocol. For example, Bluetooth Low Energy is very strict and predictable in its schedule of radio operations; advertisement and connection intervals occur at set times. In contrast, an 802.15.4 protocol is more flexible in the timing of many message events; CSMA (carrier sense multiple access) in IEEE 802.15.4 adds random backoffs so that event delays are on the order of milliseconds. This allows IEEE 802.15.4 messages to be sent around the Bluetooth Low Energy events and still be reliably received.

4.2 802.15.4 RAIL Priority

802.15.4 protocols currently have three RAIL priorities.

No.	Name	Default Setting	Exit Criterion
1	Active TX	100	MAC ACK received (or not)
2	Active RX	255	Packet filtered or MAC ACK sent
3	Background RX	255	Task with higher Priority present

If an Active TX gets executed the radio will be released at the time the corresponding MAC acknowledgement was received (or a timeout occurred).

Background RX will leave the radio in receive state ready to receive asynchronous messages. If the active RX priority is different than the background RX priority, the receive priority will be raised whenever a sync word is detected and only lowered once that packet is filtered or completed and its ACK is sent if one was requested.

4.2.1 Balancing Priorities

As explained in section 6.1 Bluetooth Priorities, by default the Bluetooth priority range is mapped into the RAIL priority range 16 - 32. In general, Bluetooth starts out using low priority (32) and dynamically increases the priority up to the maximum (16) as needed if messages are not succeeding.

As described in the previous section, an 802.15.4-based stack such as Zigbee or Connect uses default RAIL priority values of 255 for background RX, 255 for active RX, and 100 for active TX.

As a result of these default RAIL priorities, in an 802.15.4 protocol-Bluetooth multiprotocol application, by default Bluetooth traffic will always take priority over 802.15.4 protocol traffic. This is a good choice for many applications, because Bluetooth traffic has stringent timing requirements, unlike 802.15.4 protocols. However, if Bluetooth traffic load is very high (for example, sending lots of data using a very small connection interval), it is possible for 802.15.4 protocol traffic to be completely blocked from access to the radio because of its lower priority and the very small windows of available radio time left by the Bluetooth traffic.

Note: The following information is currently only applicable to the EmberZNet Zigbee stack. Silicon Labs Connect does not yet have the API needed to change the priorities.

If you are developing an 802.15.4-based dynamic multiprotocol application, and it is important for that traffic to succeed in the presence of very high load Bluetooth traffic, you can adjust the default priorities as shown in the table below using the following API:

```
EmberStatus emberRadioSetSchedulerPriorities(const EmberMultiprotocolPriorities *priorities)
```

No.	Name	Default Setting
1	Active TX	23
2	Active RX	24
3	Background RX	255

Because the Bluetooth initially sets its RAIL priority to 32, these 802.15.4 priority settings give 802.15.4 traffic higher priority than Bluetooth initially, which gives the 802.15.4 protocol a chance to transmit or receive traffic successfully even in the presence of a very high load of Bluetooth traffic. On the other hand, Bluetooth will dynamically increase its priority if it is bumped from the scheduler by the 802.15.4 traffic, up to a high priority of 16. Thus, after allowing the 802.15.4 protocol access to the radio initially, Bluetooth will take priority on subsequent retries if necessary.

This approach allows both protocols to compromise on their use of the radio without one being able to completely dominate over the other.

5. Implementing Multiprotocol with RAIL

This chapter offers more information about the particularities of RAIL for users who consume the RAIL API directly to develop propriet-ary protocols. In particular it offers details on how to work with the RAIL APIs to handle specific radio scheduler cases.

5.1 Examples with Background Receive, Yield Radio and State Transition

The fundamentals of the RAIL Multiprotocol priority system are straightforward: a radio event with a higher priority (that is, smaller in number) will always usurp any other radio events with lower priority. However, this topic becomes more complicated when considering state transitions and APIs such as `RAIL_StartRx()`, which put the radio into a certain state for an indefinite amount of time. This section provides some illustrations and examples to demonstrate how these time-unbounded states are handled, and how the application layer can use APIs such as `RAIL_YieldRadio()` to control them. The examples are as follows:

- State Transitions with a Single Protocol
- State Transitions with Two Protocols
- State Transitions with Two Protocols and Monotonically Increasing Priorities

In these examples, `RAIL_StartTx()` is the source of the TX event that interrupts the background RX. Note, however, that these examples are applicable to any radio API except for `RAIL_StartRx()`. In other words, the examples are applicable to any API that starts a radio event that is not a background RX.

These examples illustrate expected multiprotocol behaviors regarding state transitions. To summarize:

- In a state transition, the new state is treated as an indefinite extension of the originating event at that same priority until `RAIL_YieldRadio()` is called.
- Background RX events are not affected by `RAIL_YieldRadio()`. Only `RAIL_Idle()` can permanently remove a protocol from the background RX state.
- An event with a higher priority will always usurp an event with lower priority, regardless of any other API calls.
- Only `RAIL_StartRx()` receives can be 'returned to' from a higher priority event through `RAIL_YieldRadio()` or `RAIL_Idle()`.
- All radio events other than `RAIL_StartRx()` require `RAIL_YieldRadio()` in order to end and progress to the next event.
- The call to `RAIL_YieldRadio()` cannot be replaced with `RAIL_Idle()`. `RAIL_Idle()` clears out *all* events for the given protocol.

5.1.1 State Transitions with a Single Protocol

This first example examines the behavior of the radio with a single protocol (that is, where the same `RAIL_Handle_t` is used for all radio function calls). The radio starts in RX with an initial call to `RAIL_StartRx()`, then moves into a TX with a higher priority call to `RAIL_StartTx()`. It is important to note that after the transmit is done, the radio transitions to the state specified by `RAIL_SetTxTransitions()`, and it stays in the state indefinitely at the same priority and channel as the TX until `RAIL_YieldRadio()` is called. After that, the radio returns to RX, with the initially specified priority and channel.
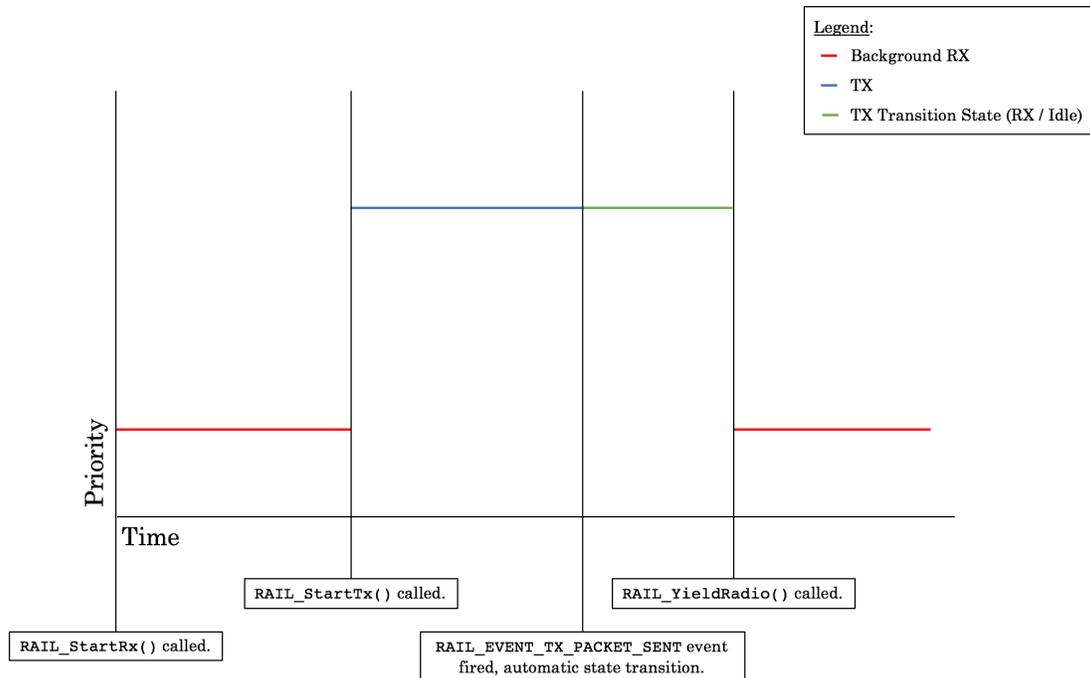
Figure 5.1. State Transitions with Calls to `RAIL_StartTx()`, `RAIL_StartRx()`, `RAIL_YieldRadio()` with a Single Protocol

The need to actively yield the radio, and thus the `RAIL_YieldRadio()` API were necessary largely due to ACK'ing. The design philosophy is that, because both a TX and a received ACK are viewed as part of the same transaction, if a node transmits and expects an ACK it should be able to both transition to RX and continue listening for the ACK as part of the same operation (and therefore same priority) as the original TX. In general, however, RAIL on its own cannot know whether or not an ACK is required. This can depend on other factors, such as packet contents, or other application logic, and so cannot be simply determined by checking whether ACK'ing has been configured with `RAIL_ConfigAutoAck()`.Therefore, discretion as to when a radio transaction is complete is left to the application/stack.

In the case that an ACK is not required, Silicon Labs recommends calling `RAIL_YieldRadio()` as part of handling the `RAIL_EVENT_TX_PACKET_SENT` event. Doing this causes the green line in the above figure to be minimized down to the interrupt latency time. If the application does expect an ACK, `RAIL_YieldRadio()` should be called when the ACK is received or has been deemed to time out.

5.1.2 State Transitions with Two Protocols

This scenario is similar to the first scenario regarding state transitions after TX, but introduces another protocol.
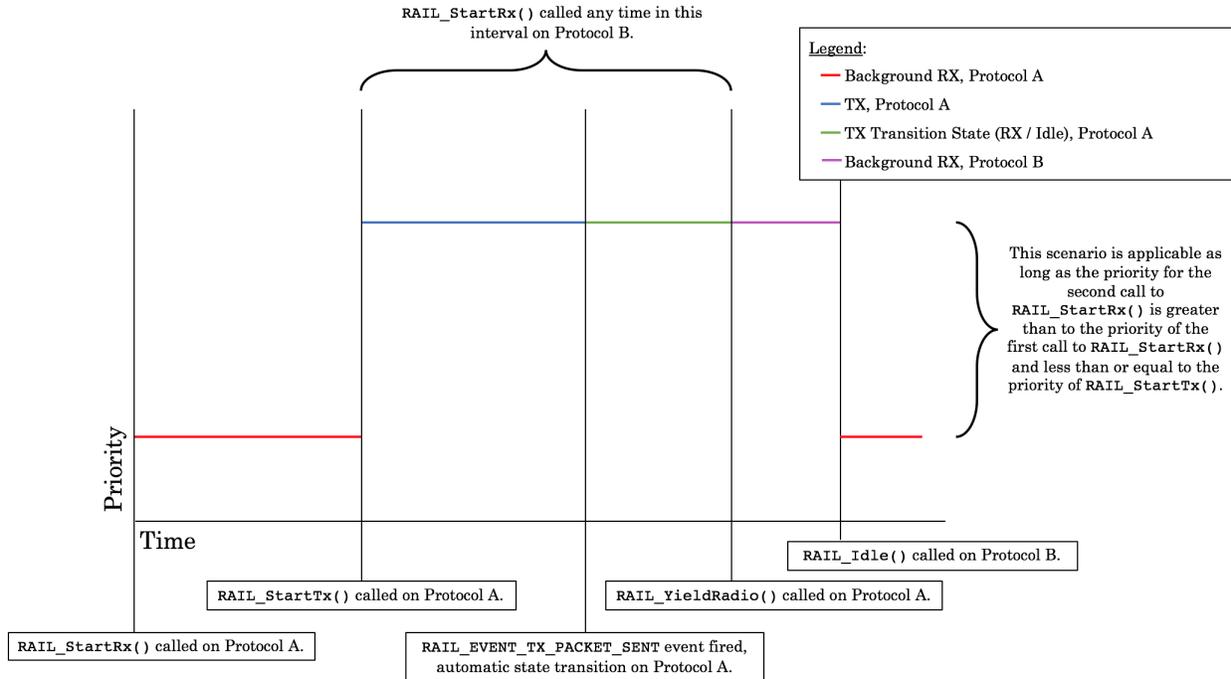
**Figure 5.2. State Transitions with Calls to `RAIL_StartTx()`, `RAIL_StartRx()`, `RAIL_YieldRadio()` With Two Protocols**

In this situation, it is important to note that `RAIL_StartRx()` can be called at any time during the TX transaction. As long as its priority is less than or equal to the priority of the TX, the RX will not come into effect until the application calls `RAIL_YieldRadio()` on Protocol A. When `RAIL_StartRx()` is called during the TX, the RX is merely added to the queue of events to be handled.

Another key point is that, although `RAIL_YieldRadio()` on Protocol A will transition from TX on Protocol A to RX on Protocol B, a `RAIL_Idle()` on Protocol B is required to transition from the RX on Protocol B to the RX on Protocol A. The philosophy here is that Background RXs can't really be yielded, since the event is never really over. The only way to exit is to stop the Background RX with a call to `RAIL_Idle()`.

5.1.3 State Transitions with Two Protocols and Monotonically Increasing Priorities

The final scenario is nearly identical to the previous one, except the call to `RAIL_StartRx()` on Protocol B is at a higher priority than the call to `RAIL_StartTx()` on Protocol A.
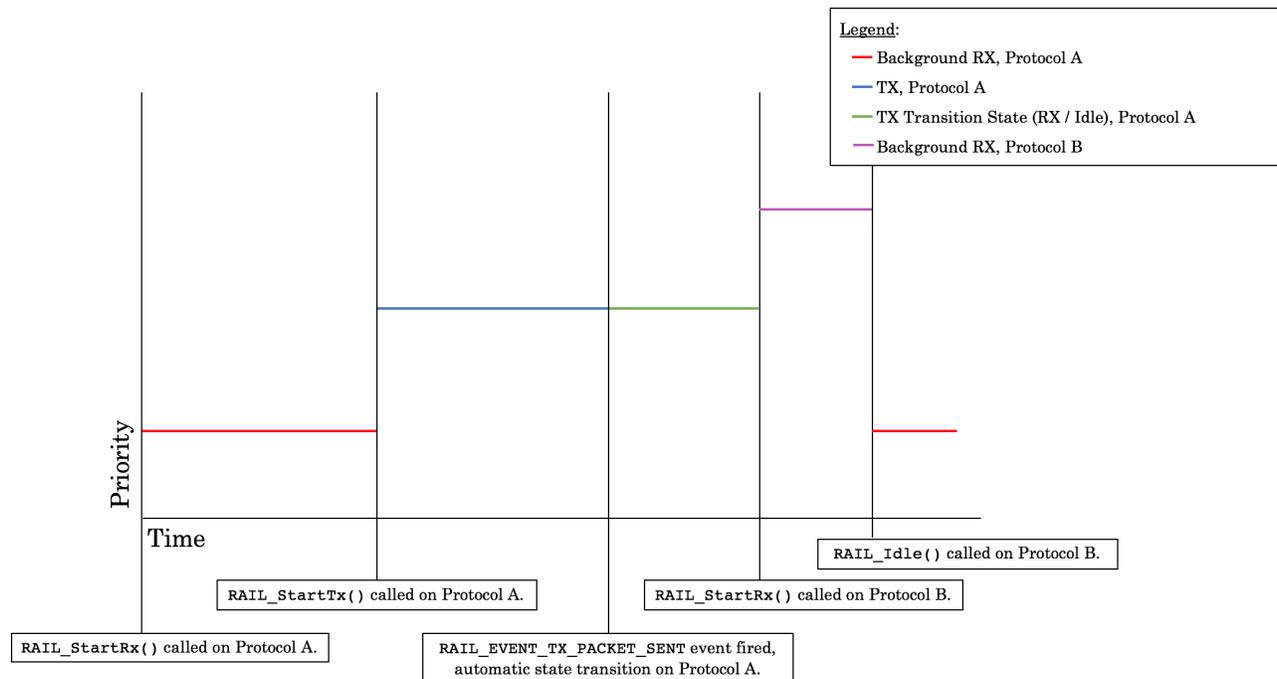
Figure 5.3. Example of State Transitions with Calls to `RAIL_StartTx()`, `RAIL_StartRx()`, `RAIL_YieldRadio()` with Two Protocols and Different Priorities

In this case, since the priority of the second `RAIL_StartRx()` is higher than the priority of the call to `RAIL_StartTx()`, a call to `RAIL_YieldRadio()` is no longer necessary. Because the second `RAIL_StartRx()` is at a higher priority, it usurps the `RAIL_StartTx()` event, taking control of the radio and removing the TX event from the state. At any time during that RX on Protocol B, `RAIL_Idle()` can be called to return to the RX on Protocol A, just as in the previous example.

Note here, that when the application calls `RAIL_Idle()` on Protocol B's RX, the application does not return to the TX Transition of Protocol A. Instead, it goes right to the background RX, even though the application never called `RAIL_Idle()` on Protocol A's TX. For Scheduled radio operations (that is, any radio operation started by an API other than `RAIL_StartRx()`), once a radio event is usurped by a higher priority event, it is removed entirely and will not be returned to later. Only Background receives, started by `RAIL_StartRx()`, can be maintained in the background and 'returned to' through a call to `RAIL_YieldRadio()` or `RAIL_Idle()`.

To emphasize the difference between `RAIL_YieldRadio()` and `RAIL_Idle()` it is important to note that, for all these examples, the call to `RAIL_YieldRadio()` cannot be replaced with `RAIL_Idle()`. `RAIL_Idle()` clears out *all* events for the given protocol – both the Background (that is, started by `RAIL_StartRx()`) and Scheduled (that is, started by APIs other than `RAIL_StartRx()`) operations. `RAIL_Idle()` would indeed still cause the application to exit out of the TX transition state, but it would also clear out the Background RX, causing the application to return to idle, not RX.

6. Implementing Multiprotocol with Bluetooth

For details on how the RAIL/Bluetooth light/switch multiprotocol example was implemented, and for more information on developing a multiprotocol application with your own protocol on RAIL, see *AN1134: Dynamic Multiprotocol Development with Bluetooth and Proprietary Protocols on RAIL in GSDK v2.x* or *AN1269 Dynamic Multiprotocol Development with Bluetooth and Proprietary Protocols on RAIL in GSDK v3.x and Higher.*

6.1 Bluetooth Priorities

As opposed to Zigbee with statically defined priorities for different operation types, Bluetooth uses a range and offset approach to assign all tasks to a given area of the priority spectrum.
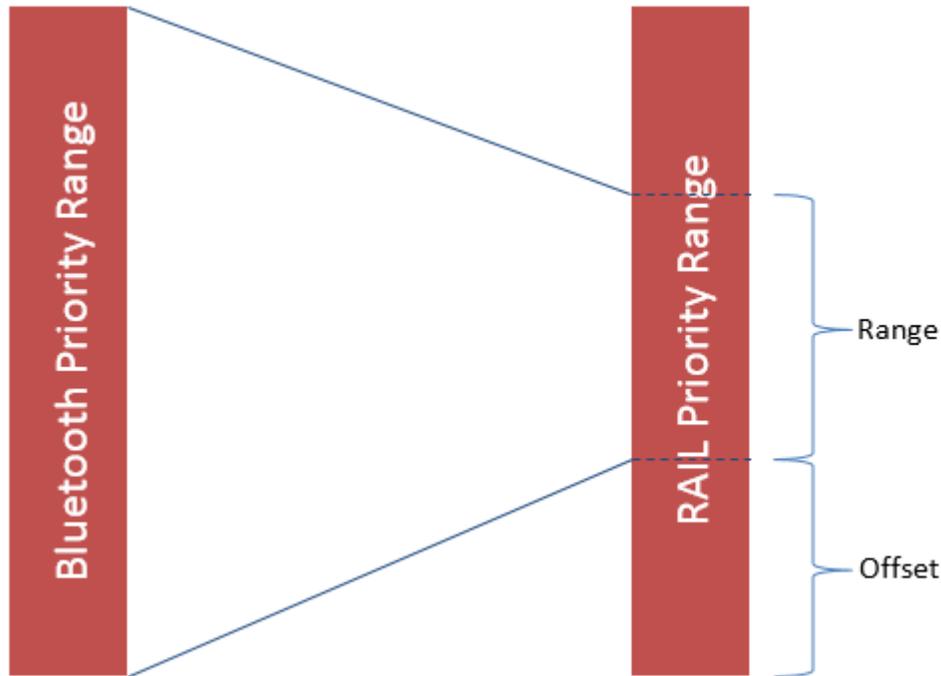
Figure 6.1. Mapping of Bluetooth Priority Range to RAIL Priority Range

In this example the Bluetooth priority range, which itself spans from 0 to 255, is mapped to a limited portion of the shared RAIL priority space.

Unlike Zigbee, Bluetooth has much more stringent timing requirements where missing a given slot may result in a connection terminating. Also Bluetooth has a range of different tasks like (potentially multiple) connections, advertisement, scanning, and Periodic Advertising with Responses (PAwR) transmissions and receptions.

Table 6.1. Different Priorities in Bluetooth

No.	Name	Default Setting	Exit Criterion
1	Connection	135 to 0	Connection Event Ends
2	Connection Initiation	55 to 15	Initiation Window Ends
3	Advertisement	175 to 127	Advertisement Event Ends
4	Scanner	191 to 143	Scan Window Ends
5	PAwR TX	15 to 5	Advertiser: PAwR Response Slot Delay Ends Synchronizer: PAwR Response Slot Ends
6	PAwR RX	20 to 10	Advertiser: PAwR Response Slot Ends Synchronizer: PAwR Response Slot Delay Ends

In order to handle this the Bluetooth scheduler, whose priorities are mapped to the RAIL radio scheduler, takes into account the following parameters for each task:

1. Start Time
2. Minimum time
3. Maximum time
4. Priority
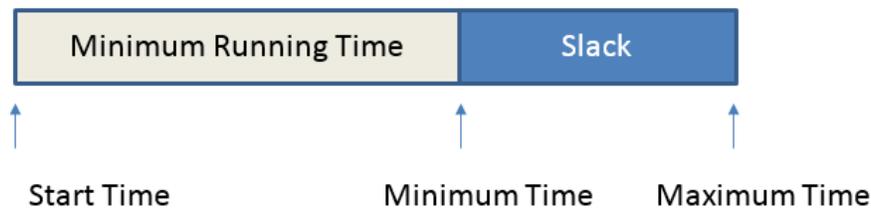
Figure 6.2. Bluetooth Task

If the start time is moved the total running time is reduced respectively, that is the slack is reduced. Also priorities can be dynamically adjusted.

6.1.1 Connections

Connections have a relatively high priority. The start time of a connection cannot be moved.

The priority is dynamically increased by the Bluetooth scheduler the closer the connection gets to the supervision timeout, and reaches the maximum priority close to it. A TX packet in the TX queue also increases the priority of a connection.

6.1.2 Connection Initiation

Connection initiation scans advertisements from target device to establish a connection. It has a higher priority compared to a scanner to allow more robust connection establishment.

6.1.3 Advertisements

Advertisements by default have a lower priority and their start point can be moved. Start time and Maximum time are defined by the advertisement interval.

If an advertisement could not be sent out, the priority of advertisements increases slowly and is reset once an advertisement was successfully sent.

6.1.4 Scanner

By default, these tasks have the lowest priority. Start, minimum and maximum time are defined by the scanning interval and window size. Scanning can continue even when interrupted by a higher priority task. If this happens the scan time is accumulated to make sure the desired scan window size is reached at each scanning interval.

As with advertisements, the priority is increased in case the desired scan interval or window size could not be previously met. It is reset back to its initial priority once the scan interval or window size has been met.

6.1.5 Periodic Advertising with Responses (PAwR)

Sending Periodic Advertising with Responses has the highest priority by default over all other Bluetooth tasks, followed by receiving responses in PAwR to maintain synchronization in an electronic shelf label (ESL) network.

A PAwR task priority is increased if the task scheduling fails twice in a row. The priority is either increased by 1/6th of the priority range, or at least by one until the maximum priority has been reached. The task priority is reset back to the minimum after successful scheduling. The same procedure applies to both PAwR advertiser and synchronizer in both directions.

6.2 Example of Bluetooth Scheduler Operation

This example illustrates how the Bluetooth scheduler will schedule three connection tasks and one advertisement task, each holding different priorities. In the following figures the gray part indicates the minimum runtime a task requires and the blue part indicates the maximum runtime the task can use and, if flexible, the region where the task can be moved. The following figure shows the initial setup.
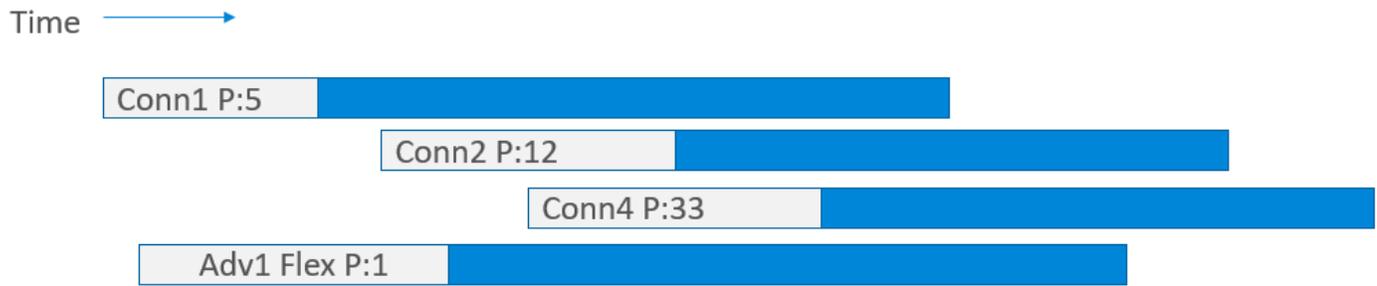
Figure 6.3. Task Scheduling Example: Setup

As shown below Conn1 is the first task to run as it does not overlap with any higher priority task.
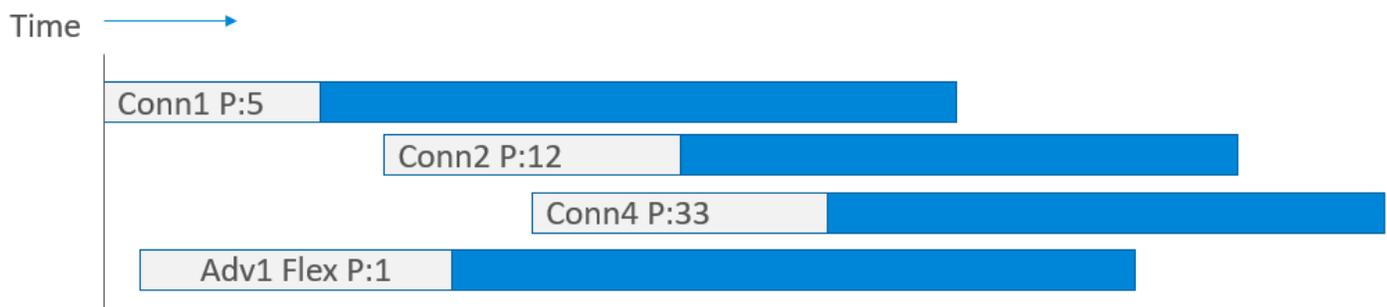
Figure 6.4. Task Scheduling Example: 1st Step

Adv1 overlaps with the higher priority Conn2. Adv1 is flexible and therefore gets moved in as illustrated in the following figure.
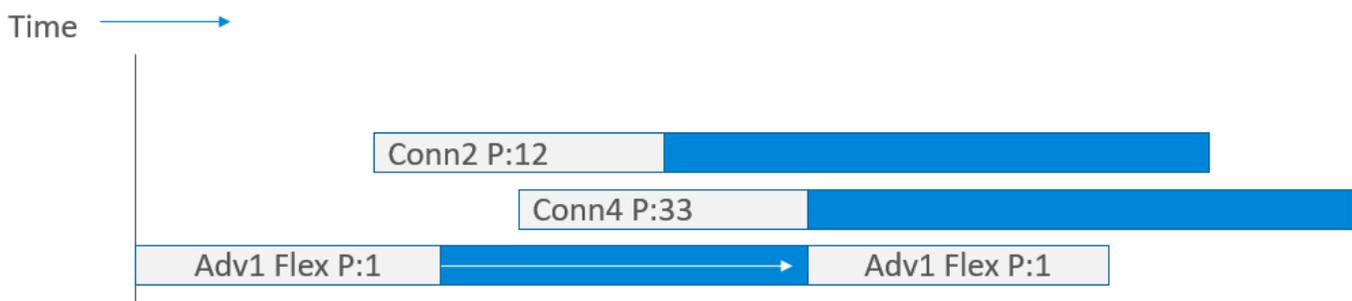
Figure 6.5. Task Scheduling Example: 2nd Step

Conn2 overlaps with higher priority task Conn4. As Conn2 is not flexible the scheduling of Conn2 fails.
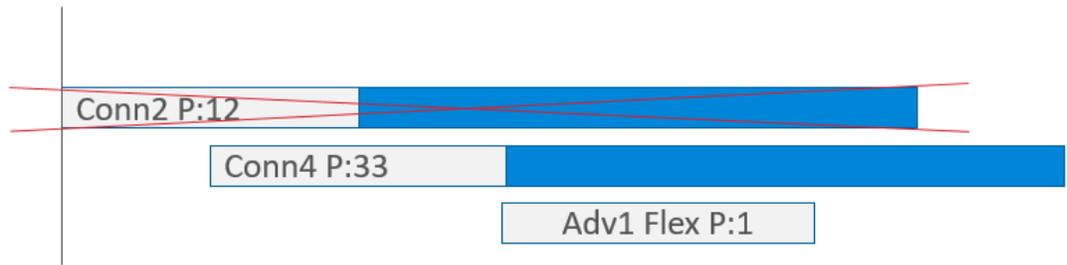
Figure 6.6. Task Scheduling Example: 3rd Step

Conn4 does not overlap with other tasks, therefore Conn1 end is adjusted to stop before Conn4 starts.
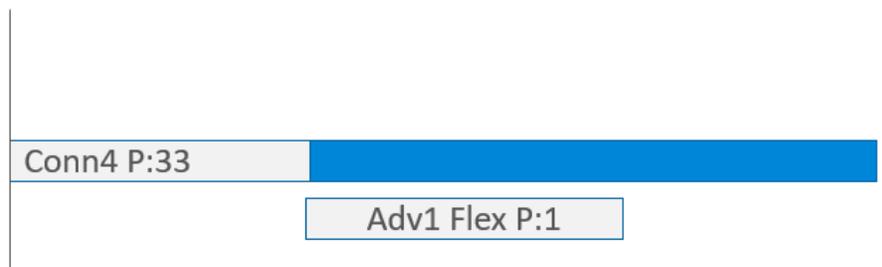
Figure 6.7. Task Scheduling Example: 4th Step

Finally Adv1 is run. Conn4 is adjusted to end before Adv1 starts
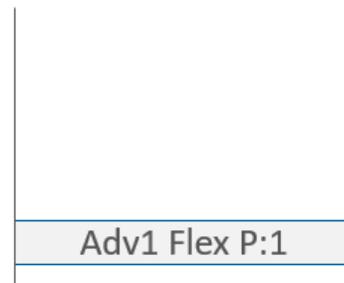
Figure 6.8. Task Scheduling Example: 4th Step

6.3 Modifying Priorities

The "sl_bt_configuration_t" (v3.x)/"gecko_configuration_t" (v2.x) struct defines the sl_bt_stack_config_t struct, which contains the field "bluetooth.linklayer_priorities" that is a pointer to the priority configuration. If the pointer is NULL then the stack uses its default priorities as listed in section 6.1 Bluetooth Priorities above as well as this section.

In case the pointer is not null it must point to a struct of priority settings as defined below:

```
typedef struct {
  uint8_t scan_min;
  uint8_t scan_max;
  uint8_t adv_min;
  uint8_t adv_max;
  uint8_t conn_min;
  uint8_t conn_max;
  uint8_t init_min;
  uint8_t init_max;
  uint8_t rail_mapping_offset;
  uint8_t rail_mapping_range;
  uint8_t _reserved;
  uint8_t adv_step;
  uint8_t scan_step;
  uint8_t pawr_tx_min;
  uint8_t pawr_tx_max;
  uint8_t pawr_rx_min;
  uint8_t pawr_rx_max;
} sl_bt_bluetooth_ll_priorities;
```

The parameters `scan_min`, `can_max`, `adv_min`, `adv_max`, `conn_min`, `conn_max`, `init_min` and `init_max` define the minimum and maximum priorities for scanning, advertisement, connections, and initiations respectively. The priorities will move between the min and max boundaries as described in sections 6.1.1 Connections to 6.1.4 Scanner above.

The RAIL mapping parameters, `rail_mapping_offset` and `rail_mapping_range`, define how the Bluetooth link layer priorities are mapped to the global RAIL radio scheduler priorities. The mapping of these values can be seen in 6.1 Bluetooth Priorities. The default for both `rail_mapping_offset` and `rail_mapping_range` is 16.

The `adv_step` and `scan_step` parameters define the step size when the priority of scanning and advertising is changed dynamically.

Finally, the parameters `pawr_tx_min`, `pawr_tx_max`, `pawr_rx_min`, and `pawr_rx_max` define the priority range for the PAwR advertiser and synchronizer TX and RX events in each subevent.

Smart. Connected.
Energy-Friendly.

IoT Portfolio
www.silabs.com/products

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

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

SILICON LABS

www.silabs.com