Transceiver Facility PIM Specification

Transceiver Facility
where software defines the radio

Document WINNF-TS-0008

Version V2.1.1

22 January 2022

TERMS, CONDITIONS & NOTICES

Table of Contents

List of Figures

List of Tables

Contributors

The following individuals and their organization of affiliation are credited as Contributors to development of the specification, for having been involved in the work group that developed the draft then approved by WInnForum member organizations:

- Marc Adrat, FKIE,
- Claude Bélisle, NordiaSoft,
- Eric Campbell, Harris Corporation,
- Jean-Philippe Delahaye, DGA,
- Antonio Di Rocco, Leonardo,
- David Hagood, Cobham,
- Frédéric Leroy, ENSTA,
- Sarah Miller, Rockwell-Collins,
- David Murotake, Hitachi Kokusai Electric,
- Eric Nicollet, Thales Communications & Security,
- Peter Troll, Rohde & Schwarz,
- Dmitri Zvernick, NordiaSoft.

Standard history

Version	Date	Contents
V1.0.0 (deprecated)	28 January 2009	Initial release.
V2.0.0	9 November 2017	Release of the *PIM specification* for V2.0.0. Major upgrade of V1.0.0.
V2.1.0	20 January 2022	Addition of the *Native C++ PSM specification*, *SCA PSM specification* and *FPGA PSM specification*. The *PIM specification* content is identical to V2.0.0.

Transceiver Facility PIM Specification

1 Introduction

This document WINNF-TS-0008-V2.1.0 is the *PIM specification* (Platform-Independent Model) of WInnForum *transceiver facility* V2.1.0.

The *transceiver facility* V2.1.0 is also composed of the following detached appendices:

- WINNF-TS-0008-App01-V2.1.0 *Transceiver Facility Native C++ PSM specification*,
- WINNF-TS-0008-App02-V2.1.0 *Transceiver Facility SCA PSM specification*,
- WINNF-TS-0008-App03-V2.1.0 *Transceiver Facility FPGA PSM specification,*
- WINNF-TS-0008-App04-V2.1.0 *Transceiver Facility Absolute Time Use Case*.

V2.1.0 complements V2.0.0 with three *PSM specifications*.

The *Transceiver Facility* standardizes a service-oriented *Transceiver Application Programming Interface* (*Transceiver API*) and associated *Transceiver Properties*, in support of portability of *radio applications* and hospitality of reconfigurable *transceiver* implementations.

The *transceiver* is the processing stage situated between the antenna and the radio physical layer baseband processing. Its I/O signals are the *baseband signal* and the *radio signal* (see section 1.2.2), as depicted in following figure:

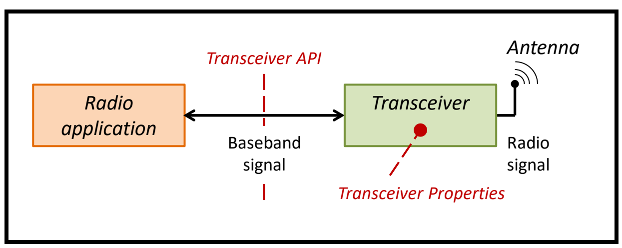

Figure 1 Overview of Transceiver Facility

1.1 Specification approach

1.1.1 Model Driven Architecture (MDA)

The *Transceiver Facility* structure is inspired by application of the Object Management Group (OMG) Model Driven Architecture (MDA) approach (see [Ref1]) to the technical domain of physical layer engineering of software-defined radio (SDR) systems.

The *Transceiver Facility* is composed of a *core specification*, denoted as the *Platform-Independent Model (PIM) specification* (this document) and *appendices*.

The *core specification* answers to the definition of a PIM provided by [Ref1]: "*A PIM exhibits a sufficient degree of independence so as to enable its mapping to one or more platforms. This is commonly achieved by defining a set of services in a way that abstracts out technical details. Other models then specify a realization of these services in a platform specific manner.*".

Appendices are *Platform-Specific Model (PSM) specifications* specified for a number of programming paradigms supporting implementation of the PIM software interfaces.

The *PSM specifications* answer to the definition of a PSM provided by [Ref1]: "*A PSM combines the specifications in the PIM with the details required to stipulate how a system uses a particular type of platform. If the PSM does not include all of the details necessary to produce an implementation of that platform it is considered abstract (meaning that it relies on other explicit or implicit models which do contain the necessary details)*.".

When no standard *PSM specification* is applicable, a *non-standard PSM* has to be formally specified through a specification structured like *standard PSMs*.

1.1.2 Implementation feedback collection

Users of a *core specification* and standard *PSM specifications* are invited to submit implementation feedback to the WInnForum for consideration in perspective improvement of the *Transceiver Facility*.

Users of a non-standard *PSM specification* are invited to submit the non-standard specification as an input document to the WInnForum to be considered for future inclusion in the *Transceiver Facility*.

1.1.3 Conventions

The *PIM specification* refers itself as "the *specification*" in the remaining of the document.

A *normative clause* of the *specification* is a particular sentence that can be:

- A *definition*: defines a general concept, contains "**is/are defined as**"; name of the defined concept is formatted in *italics*,
- A *declaration*: specifies a formal concept (e.g. a state, an interface, an error), contains "**is/are specified as/by**"; name of the declared concept is formatted according to its nature,
- A *requirement*: specifies a condition to be respected by a *transceiver*, contains "**shall**".

The term "*unspecified*" indicates an aspect that is not specified by the specification, more specific aspects being left to user's decisions.

1.1.4 Document structure

The *PIM specification* is structured as follows:

- Section 1, *Introduction*: defines essential aspects, provides an overview of the specified services groups,
- Section 2, *Services*: specifies states machines, API services groups, provide and use services,
- Section 3, *Service Primitives and Attributes*: specifies API primitives, exceptions, attributes and types,
- Section 4, *Properties*: specifies *properties* characterizing *transceiver instances*,
- Section 5, *PSM specifications*: specifies rules pertaining to derived *PSM specifications*.

1.2 Transceiver concepts

A *transceiver* **is defined as** a subsystem of a radio platform that transforms, when it transmits, baseband signal(s) into radio signal(s) and, when it receives, radio signal(s) into baseband signal(s).

A *transceiver instance* **is defined as** one particular implementation of a *transceiver*.

One or several *transceiver instances* can be available on a *radio platform* and one or several *transceiver instances* can be used by a *radio application*.

The remainder of the *specification* is applicable to any particular *transceiver instance*, assumed fully independent of any other *transceiver instance* eventually available on a given *radio platform*.

1.2.1 Channels

1.2.1.1 Tx channels

A *transmit channel* (*Tx channel*) **is defined as** an elementary part of a *transceiver instance* that transforms, when it transmits, one *baseband signal* (see section 1.2.2.1) into one *radio signal* (see section 1.2.2.2).

A *transmission* **is defined as** a phase during which a *Tx channel* continuously transmits.

Up-conversion **is defined as** the signal processing performed by a *Tx channel* during a *transmission*.

A *transceiver instance* can have zero to several *Tx channels*. All *Tx channels* of a specific *transceiver instance* are controlled simultaneously by the *radio application* and operate synchronously.

TX_CHANNELS (see section 4.2) specifies the number of *Tx channels* of a *transceiver instance*.

1.2.1.2 Rx channels

A *receive channel* (*Rx channel*) **is defined as** an elementary part of a *transceiver instance* that transforms, when it receives, one *radio signal* into one *baseband signal*.

A *reception* **is defined as** a phase during which an *Rx channel* continuously receives.

Down-conversion **is defined as** the signal processing performed by an *Rx channel* during a *reception*.

A *transceiver instance* can have zero to several *Rx channels*. All *Rx channels* of a specific *transceiver instance* are controlled simultaneously by the *radio application* and operate synchronously.

RX_CHANNELS (see section 4.2) specifies the number of *Rx channels* of a *transceiver instance*.

1.2.1.3 Transceiver categories

A *simplex transceiver* **is defined as** a *transceiver* with transmit or receive capability, but not both. A *simplex transceiver* has one or many *Tx channels*, or one or many *Rx channels*.

A *duplex transceiver* **is defined as** a *transceiver* with one or many *Tx channels* and one or many *Rx channels*.

A *full-duplex transceiver* **is defined as** a *duplex transceiver* which *transmission* and *reception* phases are fully independent and can occur simultaneously.

A *half-duplex transceiver* **is defined as** a *duplex transceiver* with no simultaneous *transmission* and *reception* phases, due to sharing of critical processing resources between its *Tx channels* and *Rx channels*.

DUPLEX (see section 4.2) specifies if a *duplex transceiver* is *half-duplex* or *full-duplex*.

1.2.2 I/O signals

1.2.2.1 Baseband signal

A *baseband signal* ($\underline{s}_{BB}$) **is defined as** the complex digital signal exchanged between a *radio application* and *Tx channels* or *Rx channels*.

The *baseband sampling frequency* (F_s^{BB}) **is defined as** the sampling frequency of a *baseband signal*.

A *baseband sample* ($\underline{s}_{BB}[n]$) **is defined as** a complex sample of the *baseband signal*, with $\underline{s}_{BB}[n] = I + i.Q$, where $i = \sqrt{-1}$.

The *in-phase component* (*I*) of a *baseband sample* **is defined as** its real part.

The *quadrature component* (*Q*) of a *baseband sample* **is defined as** its imaginary part.

$\dot{\underline{s}}_{BB}$ **is defined as** the Fourier transform of $\underline{s}_{BB}$.

L_{BB} **is defined as** the level of the *baseband signal* expressed in decibels relative to full scale (dBFS) for the applied numerical representation.

The *full-scale* (FS) of the numerical representation of the *baseband signal* **is specified as**, depending on value of **IQ_TYPE** (see section 4.5):

- `2^15-1` if `IQ_TYPE` is equal to *16bit*,
- `2^31-1` if `IQ_TYPE` is equal to *32bit*,
- `1.0` if `IQ_TYPE` is equal to *floatingPoint*.

L_{BB} **shall** be computed according to $L_{BB} = 10.\log_{10}\left(\frac{\frac{1}{N}\sum_{i=0}^{i=N-1}|s_{BB}[n_0+i]|^2}{FS^2}\right)$.

1.2.2.2 Radio signal

The *radio signal* (s_{RF}) **is defined as** the analogue voltage signal at the output of *Tx channel*, during a *Transmission*, or at the input of *Rx channel*, during a *Reception*.

Radio signal is typically taken at the antenna connector, but can be defined elsewhere depending on usage context.

The *carrier frequency* (f_c) **is defined as** the radio frequency around which the *radio signal* spectrum is positioned.

Note: the *carrier frequency* is the center frequency of the *Tx signal* measured spectrum when the *baseband signal* is symmetrical. It is not always the case, e.g. in the case of single side band modulations.

$\dot{s}_{RF}$ **is defined as** the Fourier transform of s_{RF}.

L_{RF} **is defined as** the level of the *radio signal* expressed in decibels relative to one milliwatt (dBm).

1.2.3 Processing phases

A *processing phase* **is defined as** a continuous period of time during which *Rx channels* or *Tx channels* perform a signal processing transformation.

The *activation time* of a *processing phase* **is defined as** the time at which the *processing phase* starts.

The *termination time* of a *processing phase* **is defined as** the time at which the *processing phase* stops.

A *baseband block* **is defined as** the *baseband signal* exchanged between a *radio application* and one *Rx channel* or one *Tx channel* during a *processing phase*.

The *sample number* of a *baseband sample* **is defined as** its position within a *baseband block*, starting at `1` for the first sample.

1.2.4 Transmission

A *transmission* **is defined as** the *processing phase* of *Tx channels*.

The following figure illustrates the principle of a *transmission*:

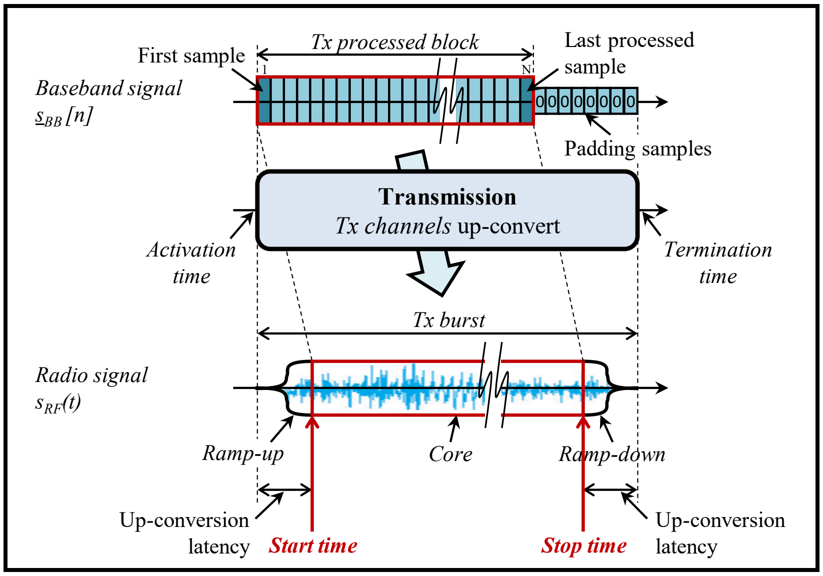

Figure 2 Principle of transmission processing phase

1.2.4.1 Boundary signals

A *transmit forwarded block* (*Tx forwarded block*) **is defined as** a the *baseband block* sent by a *radio application* to one *Tx channel* during a *transmission*.

A *transmit packet* (*Tx packet*) **is defined as** a one elementary set of *baseband samples* successively sent by a *radio application* to one *Tx channel* for transfer of a *Tx forwarded block*.

A *transmit processed block* (*Tx processed block*) **is defined as** a the part of the *Tx forwarded block* up-converted by one *Tx channel* during a *transmission*.

Correct operation of *Tx channel*s requires that the level of *baseband signal* is within a particular range.

The upper bound of this range generally corresponds to the maximum level maintaining *Tx channels* linearity. The lower bound of this range generally corresponds to the level required for the *baseband signal* to be able to drive the *Tx channels* processing.

A *transmit burst* (*Tx burst*) **is defined** as the *radio signal* sent by one *Tx channel* to the *antenna* during a *transmission*.

The *core* of a *Tx burst* **is defined as** the part of the *Tx burst* without its *ramp-up* and *ramp-down*.

1.2.4.2 Start and stop times

The *start time* of a *Tx burst* **is defined as** the start time of its *core*.

The *start time* of a *Tx burst* generally happens *up-conversion* latency after *activation time*.

The *stop time* of a *Tx burst* **is defined as** the stop time of its *core*.

The *stop time* of a *Tx burst* generally happens *up-conversion* latency before *termination time*.

1.2.4.3 Transmit transfer function

An ideal *up-conversion* generates a *radio signal* which spectrum is the zero-centered spectrum of the *baseband signal* translated around the *carrier frequency*, with application of an ideal low-pass filter of bandwidth B to select the spectrum portion of interest.

An ideal *up-conversion* obeys to the following equation:

$$\boxed{\underline{\dot{s}}_{RF}(f+f_c) = \alpha.\mathrm{rect}(f/B).\underline{\dot{s}}_{BB}(f),\ f \in [-F_S^{BB}/2\,;+F_S^{BB}/2]} \qquad \text{Eq. 1,}$$

where:

- rect( ) is the rectangular function,
- α is a real coefficient reflecting the *up-conversion* gain.

The *transmit transfer function* ($\underline{H}_{Tx}$) **is defined as** the transfer function nearing the ideal low-pass filter of the ideal *up-conversion* that is implemented by a *Tx channel*.

CHANNEL_MASK (see section 4.10) specify the frequency domain mask into which $\underline{H}_{Tx}$ fits.

The actual *up-conversion* performed by a *Tx channel* obeys to the *up-conversion formula*:

$$\boxed{\underline{\dot{s}}_{RF}(f+f_c) = \underline{H}_{Tx}(f).\underline{\dot{s}}_{BB}(f),\ f \in [-F_S^{BB}/2\,;+F_S^{BB}/2]} \qquad \text{Eq. 2.}$$

The *transmit impulse response* (h_{Tx}) **is defined as** the non-causal equivalent impulse response corresponding to *up-conversion*, symmetrical around the y-axis, with *up-conversion latency* equal to the half of its domain:

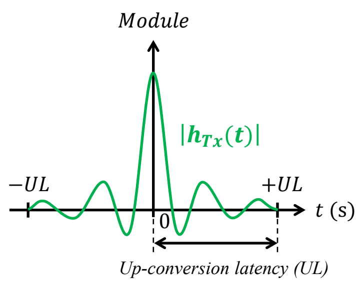

Figure 3 Transmit impulse response

One has:

$$\boxed{\begin{array}{c} s_{RF}(t) = \sum_{k=0}^{L-1}\left[\left(\Re\left(\underline{s}_{BB}[k]\right).\cos(2\pi f_c t) - \Im\left(\underline{s}_{BB}[k]\right).\sin(2\pi f_c t)\right).h_{Tx}(t-t_s-k/F_S^{BB})\right], \\ t \in [t_s; t_s + L/F_S^{BB}] \end{array}}$$

Eq. 3,

where:

- L denotes the *transmit block length*,
- $\Re(\ )$ and $\Im(\ )$ denote the real and imaginary part of a complex number,
- t_s denotes the *start time*.

Further technical information is available in technical literature, e.g. [Ref2] and [Ref3].

1.2.4.4 Transmit gain

The *transmit gain* (G_{Tx}) of a *transmission* **is specified as** $G_{Tx} = L_{RF} - L_{BB}$.

1.2.4.5 Tx shaping

Nominal shaping **is defined as** the case where the *ramp-up* and *ramp-down* parts of the *Tx burst* are the *ramp-up* and *ramp-down* of *up-conversion*.

Ad-hoc shaping **is defined as** the case where the *ramp-up* or *ramp-down* parts of the *Tx burst* modifies the *ramp-up* and *ramp-down* of *up-conversion*.

Ad-hoc shaping is *unspecified*, and has to be specified according to the *radio application* needs.

`TX_SHAPING` (see section 4.2) specifies if the *shaping* is *nominal* or *specific*:

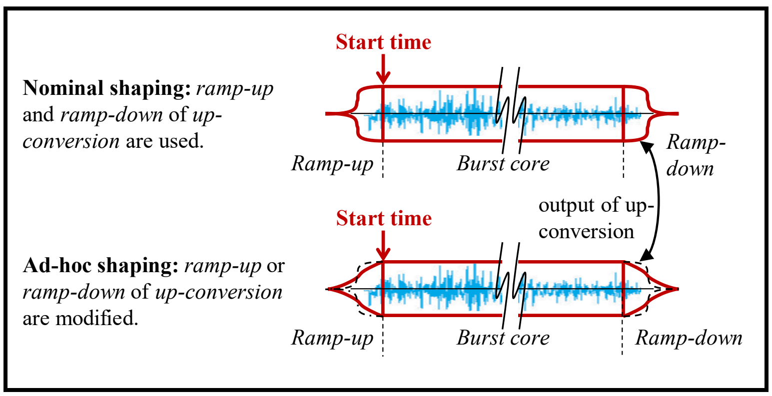

Figure 4 Nominal and specific Tx bursts shapings

1.2.5 Reception

A *reception* **is defined as** the *processing phase* of *Rx channels*.

The following figure illustrates the principle of a *reception*:

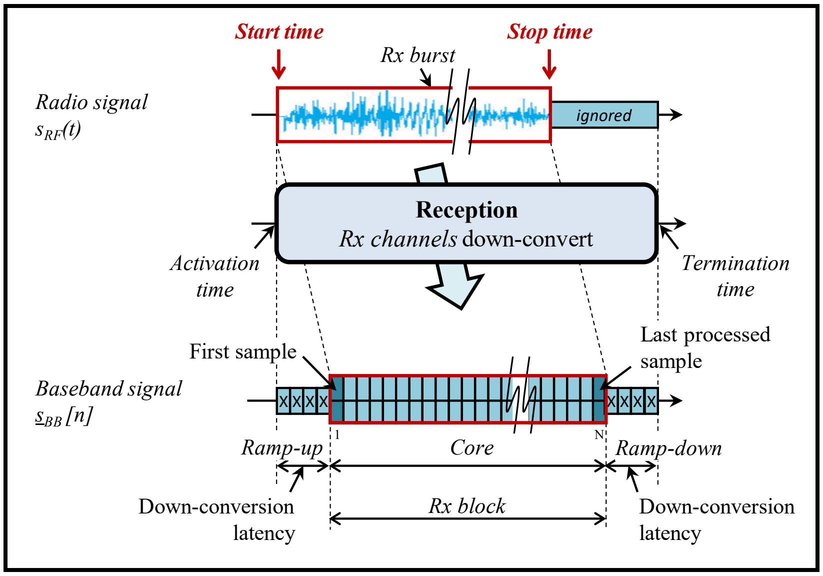

Figure 5 Principle of reception processing phase

1.2.5.1 Boundary signals

A *receive burst* (*Rx burst*) **is defined as** the *radio signal* sent by the *antenna* to one *Rx channel* during a *reception*.

Correct operation of *Rx channel*s requires that the level of *radio signal* is within a particular range.

The upper bound of this range generally corresponds to the protection of *Rx channels* against high level signals. The lower bound of this range generally corresponds to the expected sensitivity.

A *receive block* (*Rx block*) **is defined** as the *baseband block* sent by one *Rx channel* to a *radio application* during a *reception*.

A *receive packet* (*Rx packet*) **is defined as** a one elementary set of *baseband samples* successively sent by one *Rx channel* to a *radio application* for transfer of an *Rx block*.

Correct operation of a receiving *radio application* requires that the level of *baseband signal* is within a particular range.

The upper bound of this range generally corresponds to the level maximum level allowed to avoid saturation of the *radio application* processing. The lower bound of this range generally corresponds to the level under which the quantization noise impacts the reception performance.

1.2.5.2 Start and stop times

The *start time* of an *Rx burst* **is defined as** the time when the *Rx burst* starts.

The *start time* of an *Rx burst* is equal to its *activation time*.

The *stop time* of an *Rx burst* **is defined as** the time when the *Rx burst* stops.

The *stop time* of an *Rx burst* generally happens two times *down-conversion* latency before *termination time*, in order for the down-conversion processing chain to be fully flushed.

1.2.5.3 Receive transfer function

An ideal *down-conversion* generates a *baseband signal* which zero-centered spectrum is obtained from a perfect transposition of the *radio signal* spectrum considered around the *carrier frequency*, with application of an ideal low-pass filter of bandwidth B to select the spectrum portion of interest.

An ideal *down-conversion* obeys to the following equation:

$$\boxed{\underline{\dot{s}}_{BB}(f) = \alpha.\mathrm{rect}(f/B).\underline{\dot{s}}_{RF}(f-f_c),\ f \in [-F_S^{BB}/2\,;+F_S^{BB}/2]} \qquad \text{Eq. 4,}$$

where:

- rect( ) is the rectangular function,
- α is a real coefficient reflecting the *down-conversion* gain.

The *receive transfer function* ($\underline{H}_{Rx}$) **is defined as** the transfer function nearing the ideal low-pass filter of the ideal *down-conversion* that is implemented by an *Rx channel*.

CHANNEL_MASK (see section 4.10) specify the frequency domain mask into which $\underline{H}_{Rx}$ fits.

The actual *down-conversion* performed by an *Rx channel* obeys to the *down-conversion formula*:

$$\boxed{\underline{\dot{s}}_{BB}(f) = \underline{H}_{Rx}(f).\underline{\dot{s}}_{RF}(f-f_c),\ f \in [-F_S^{BB}/2\,;+F_S^{BB}/2]} \qquad \text{Eq. 5.}$$

The *receive impulse response* (h_{Rx}) **is defined as** the non-causal equivalent impulse response corresponding to *down-conversion*, symmetrical around the y-axis, with *down-conversion latency* equal to the half of its domain:

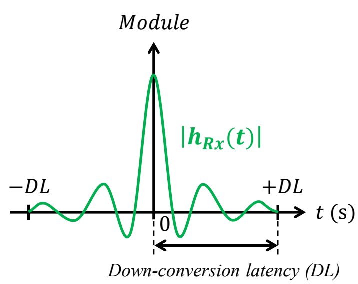

Figure 6 Receive impulse response

One has:

$$s_{BB}[k] = \left(\left(s_{RF}(t) + i.\hat{s}_{RF}(t)\right).e^{-2\pi i f_c t}\right) * h_{Tx}(t), t = t_s + k/F_s^{BB}, k \in [0; L-1] \quad \text{Eq. 6,}$$

where:

- $\hat{s}_{RF}(t)$ denotes the Hilbert transform of $s_{RF}(t)$,
- $*$ denotes the convolution product operator,
- t_s denotes the *start time*,
- L denotes the *receive block length*.

Further technical information is available in technical literature, e.g. [Ref2] and [Ref3].

1.2.5.4 Receive gain

The *receive gain* (G_{Rx}) of a *reception* **is specified as** $G_{Rx} = L_{BB} - L_{RF}$.

1.2.6 Inter-burst characterization

The *inter-burst duration* **is defined as** the duration of the period occurring between two consecutive *core bursts*.

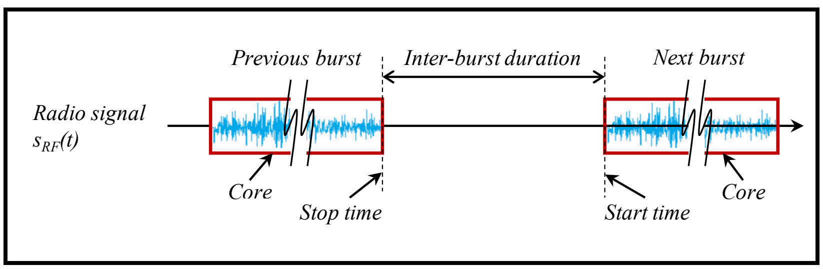

Figure 7 Principle of inter-burst duration

INTER-BURST (see section 4.8) specifies the minimum value possibly taken by *inter-burst duration*.

The *inter-processing duration* **is defined as** the duration of the period occurring between two consecutive *processing phases*, as illustrated in the following figure:

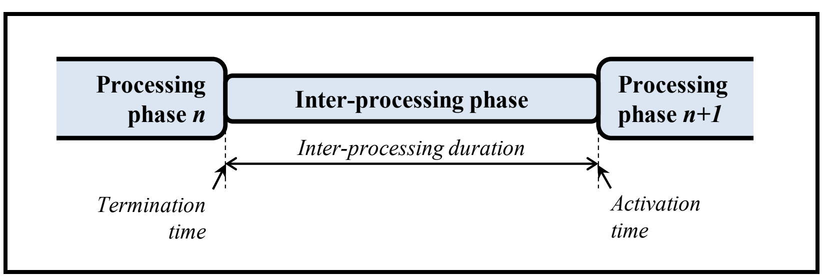

Figure 8 Principle of inter-processing duration

INTER-PROCESSING (see section 4.8) specifies the minimum value possibly taken by *inter-processing duration*.

Inter-burst duration and *inter-processing duration* is at least equal to the *tuning duration* between the two bursts.

In addition to *tuning duration, inter-burst duration* comprises the duration of the bursts ramp-up and ramp-down, while *inter-processing duration* does not.

1.2.7 Transceiver time

Transceiver time **is defined as** the *monotonic time* implemented by a *transceiver instance*, used to exchange time specification of events related to operation of the *transceiver*.

Transceiver time is essentially used in the case absolute burst creation (see section 2.4.2.3), and values of *transceiver time* can be accessed by *radio applications* using a dedicated service (see section 2.4.7).

1.3 Transceiver API

The *Transceiver API* **is defined as** the service-oriented Application Programming Interface (API) of the *specification*.

1.3.1 Services

A *service* of the *Transceiver API* **is defined as** a capability of a *transceiver instance* that exchanges messages with a *radio application* in compliance with one attached software interface and the specified behavior.

A *service interface* **is defined as** the particular Interface Description Language (IDL) software interface attached to a *service*.

A *service* and its *service interface* have the same name.

A *provide service* **is defined as** a *service* whose *service interface* is used by a *radio application* and provided by a *transceiver instance*.

A *use service* **is defined as** a *service* whose *service interface* is used by a *transceiver instance* and provided by a *radio application*.

1.3.2 Services groups

A *services group* of the *Transceiver API* **is defined as** a set of *provide services* and *use services* sharing a common purpose.

The *module* of a *service group* **is defined as** the IDL module of the *interfaces* of the *services* of the *services group*.

A *services group* and its *module* have the same name.

The following *services groups* are specified:

- Management: general control,
- BurstControl: creation and termination of *bursts*,
- BasebandSignal: packet-based exchange of *baseband blocks*,
- Tuning: control of the *tuning parameters*,
- Notifications: notification of *events* and *errors* to the *radio application*,
- GainControl: automated gain control,
- TransceiverTime: access to *transceiver time*,
- Strobing: trigger of strobes for creation of *strobed bursts*.

1.3.3 Implementation of services

An *active instance* of a *service* **is defined as** a running implementation of the *service* that is connected to the *radio application* in conformance with the *service interface*.

1.3.3.1 Access capabilities

The *transceiver instances access* capability **is defined** as the capability for the *radio application* software to access, before the CONFIGURED state is reached, to the *transceiver instances* it uses.

The *active services access* capability **is defined** as the capability for the *radio application* software to access, before the CONFIGURED state is reached, to the *active services instances* of the *transceiver instances* it uses.

The solution for *transceiver instance access* and *active services access* has to be specified by the applied *PSM specification*.

1.3.3.2 Tx channels services

SamplesTransmission (see section 2.4.2.5) is the service enabling *Tx forwarded block* exchange.

A *transceiver instance* **shall** have one *active instance* of SamplesTransmission per *Tx channel*.

This implies TX_CHANNELS instances of the *service* are implemented.

TX_SERVICES (see section 4.2) specifies, if TX_CHANNELS > 0, the set of *services* attached to *Tx channels*.

A *transceiver instance* **shall** have, for each *service* attached to *Tx channels*, one *active instance* of the *service* that jointly applies to all *Tx channels*.

1.3.3.3 Rx channels services

SamplesReception (see section 2.4.2.5) is the service enabling *Rx block* exchange.

A *transceiver instance* **shall** have one *active instance* of SamplesReception per *Rx channel*.

This implies `RX_CHANNELS` instances of the *service* are implemented.

`RX_SERVICES` (see section 4.2) specifies, if `RX_CHANNELS` > `0`, the set of *services* attached to *Rx channels*.

A *transceiver instance* **shall** have, for each *service* attached to *Rx channels*, one *active instance* of the *service* that jointly applies to all *Rx channels*.

2 Services

2.1 Provide services

The following table lists the *provide services* of the API (used by a *radio application* and provided by a *transceiver instance*, see section 1.3.1):

Services groups / Modules	Services / Interfaces	Primitives
Management	::Management::Reset	*reset()*
	::Management::RadioSilence	*startRadioSilence()* *stopRadioSilence()*
BurstControl	::BurstControl::DirectCreation	*startBurst()*
	::BurstControl::RelativeCreation	*scheduleRelativeBurst()*
	::BurstControl::AbsoluteCreation	*scheduleAbsoluteBurst()*
	::BurstControl::StrobedCreation	*scheduleStrobedBurst()*
	::BurstControl::Termination	*setBlockLength()* *stopBurst()*
BasebandSignal	::BasebandSignal::SamplesTransmission	*pushTxPacket()*
	::BasebandSignal::RxPacketsLengthControl	*setRxPacketsLength()*
Tuning	::Tuning::InitialTuning	*setTuning()*
	::Tuning::Retuning	*retune()*
GainControl	::GainControl::GainLocking	*lockGain()* *unlockGain()*
TransceiverTime	::TransceiverTime::TimeAccess	*getCurrentTime()* *getLastStartTime()*
Strobing	::Strobing::ApplicationStrobe	*triggerStrobe()*

Table 1 Provide services of Transceiver API

2.2 Use services

The following table lists the *use services* of the API (provided by a *radio application* and used by a *transceiver instance*, see section 1.3.1):

Services groups	Service / Interface	Primitives
BasebandSignal	::BasebandSignal::SamplesReception	*pushRxPacket()*
Notifications	::Notifications::Events	*notifyEvent()*
	::Notifications::Errors	*notifyError()*
GainControl	::GainControl::GainChanges	*indicateGain()*

Table 2 Use services of Transceiver API

2.3 States machines

The state machines specified in this section and their associated statecharts aim to comply with the OMG Unified Modeling Language v2.5, as specified in [Ref4].

All specified transitions are instantaneous.

Errors and exceptions handling are not modeled by the specified state machines.

2.3.1 Channels

`Channels` **is specified as** the main state machine followed by *channels* of a *transceiver instance*.

An instance of `Channels` is simultaneously followed by all *Tx channels* of a *transceiver instance*.

An instance of `Channels` is simultaneously followed by all *Rx channels* of a *transceiver instance*.

The instances of `Channels` in a *half-duplex transceiver* are not independent: if *channels* are in `TUNING` or `PROCESSING` state, the other *channels* cannot be in one of those two states.

The following figure is the statechart of `Channels` state machine:

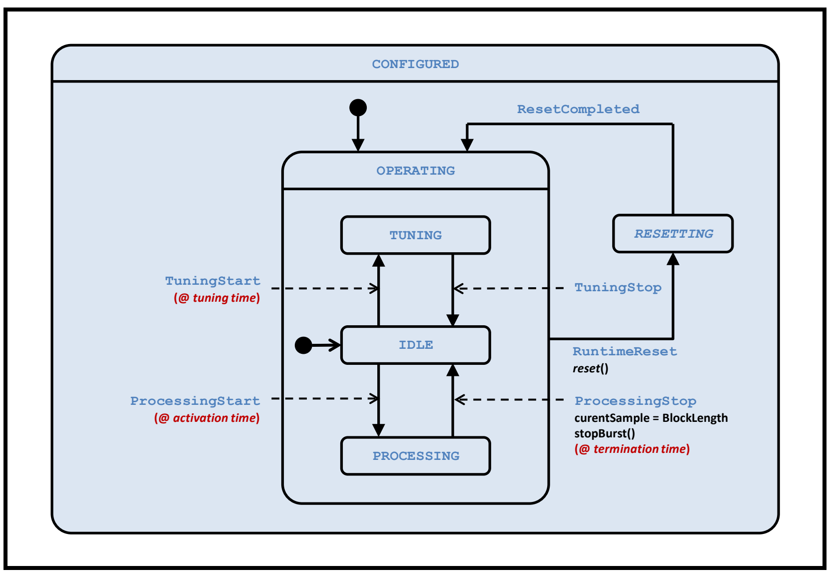

Figure 9 Channels statechart

2.3.1.1 States

2.3.1.1.1 CONFIGURED

`CONFIGURED` **is specified as** the main state of `Channels` during which *channels* of a *transceiver instance* are configured according to the needs of a supported *radio application*.

CONFIGURED is reached by the *channels* of a *transceiver instance* when they:

- Comply with the values of *properties* specified for the supported *radio application* (see section 4),
- Have attributes set to their *initial values* (see section 3.3),
- Can interact with the *radio application* according to *interfaces* of *active services*.

CONFIGURED decomposes into OPERATING and RESETTING sub-states.

Its entry transition brings to the OPERATING sub-state.

How CONFIGURED is reached has to be specified by the applied *PSM specification*.

2.3.1.1.2 OPERATING

OPERATING **is specified as** the sub-state of CONFIGURED during which *channels* are operational.

OPERATING decomposes into IDLE, TUNING and PROCESSING sub-states.

Its entry transition brings to the IDLE sub-state.

2.3.1.1.3 IDLE

IDLE **is specified as** the sub-state of OPERATING during which *channels* are inactive.

2.3.1.1.4 TUNING

TUNING **is specified as** the sub-state of OPERATING during which *channels* are tuned in accordance with the *applicable tuning parameters set*, as defined by CreationControl (see section 2.3.2).

Note: the concept of tuning of the specification is larger than only changing value of *carrier frequency*. It can imply modification of *tuning preset* and *gain*.

A *channel* **shall**, during the TUNING state, set the value of applicableTuningPreset attribute according to value of *requestedTuningPreset*:

- If equal to UndefinedTuningPreset: keep the value of applicableTuningPreset used for the *previous burst*,
- If not equal to UndefinedTuningPreset: apply *requestedTuningPreset* as the new value of applicableTuningPreset.

A *channel* **shall**, during TUNING state, set the value of applicableCarrierFreq attribute depending on value of *requestedCarrierFreq*:

- If equal to UndefinedCarrierFreq: keep the value of applicableCarrierFreq at termination of the *previous burst*,
- If not equal to UndefinedCarrierFreq: apply *requestedCarrierFreq* as the new value of applicableCarrierFreq.

A *Tx channel* **shall**, during TUNING state, set the value of applicableGain attribute depending on value of *requestedGain*:

- If equal to UndefinedGain: keep the value of applicableGain at termination of the *previous burst*,
- If not equal to UndefinedGain: apply *requestedGain* as the new value of applicableGain.

Usage of *requestedGain* by an *Rx channel* is *unspecified*.

TUNING_DURATION (see section 4.8) specifies the maximum duration of TUNING state (see section 2.3.4).

2.3.1.1.5 PROCESSING

PROCESSING **is specified as** the sub-state of OPERATING during which *channels* are in a *processing phase* (*transmission* for *Tx channels*, *reception* for *Rx channels*) (see section 1.2.3, 1.2.4 and 1.2.5).

Tx channels requirements

Tx channels **shall**, during PROCESSING state, initiate *up-conversion*:

- With *first sample* of *Tx processed block* equal to *first sample* of *Tx forwarded block*,
- With a *ramp-up signal* generated in accordance with TX_SHAPING (see section 4.2).

Tx channels **shall**, during a PROCESSING state, increment value of sampleCount (see section 3.3.1.2) each time a new *baseband sample* of the *Tx forwarded block* enters in *up-conversion*.

The *valid input level range* of *Tx channels* **is defined as** the interval [TX_MIN_BASEBAND_LEVEL ; TX_MAX_BASEBAND_LEVEL] (see section 4.10).

Tx channels **shall**, during a PROCESSING state and so long as the *baseband signal* is within the *valid input level range*, perform up-conversion in conformance with the *up-conversion formula* (see section 1.2.4).

Tx channels **shall** exhibit, during PROCESSING state and so long as the *baseband signal* level is within the *valid input level range*, an actual *gain* that belongs to applicableGain ± GAIN_ACC (see section 4.10).

Automatic level control (ALC)

Automatic level control (*ALC*) **is defined as** the capability for a *Tx channel* to automatically adjust the actually applied *transmit gain*, during early phase of the *transmission*, in order to radiate a desired level of *radio signal*.

ALC (see section 4.3) specifies the nature of the applied *ALC*.

Tx channels **shall**, during PROCESSING state and if ALC is equal to noALC, implement no *ALC*.

Tx channels **shall**, during PROCESSING state and if ALC is equal to activeALC, implement *ALC*.

Further aspects of the implemented *ALC* are *unspecified.*

Adjustment in *transmit gain* realized by an active *ALC* can be indicated to the *radio application* using the `GainControl` service (see section 2.4.6).

Rx channels requirements

Rx channels **shall**, during `PROCESSING` state, initiate *down-conversion*:

- Without transferring *ramp-up samples* to the *radio application*,
- With *first sample* of the *Rx block* equal to the sample following the *ramp-up samples*.

The *valid input level range* of *Rx channels* **is defined as** the interval `[RX_MIN_RADIO_LEVEL ; RX_MAX_RADIO_LEVEL]` (see section 4.10).

Rx channels **shall**, during a `PROCESSING` state and so long the *radio signal* is within the *valid input level range*, perform down-conversion in conformance with the *down-conversion formula* (see section 1.2.5).

Rx channels **shall**, during a `PROCESSING` state, increment value of `sampleCount` (see section 3.3.1.2) each time a new *baseband sample* generated by *down-conversion* is assigned to an *Rx packet*.

The *valid output level range* of *Rx channels* **is defined as** the interval `[RX_MIN_BASEBAND_LEVEL ; RX_MAX_BASEBAND_LEVEL]` (see section 4.10).

Rx channels **shall**, during `PROCESSING` state and so long the *radio signal* is within the *valid input level range*, deliver an output *baseband signal* which level is within the *valid output level range*.

Automatic gain control (AGC)

Automatic gain control (*AGC*) **is defined as** the capability for a *Rx channel* to automatically change the *receive gain* in order to deliver a *baseband signal* which meets the specified level requirements.

`AGC` (see section 4.3) specifies the nature of the applied AGC.

Rx channels **shall**, during `PROCESSING` state and if `AGC` is equal to `noAGC`, implement no *AGC*.

Rx channels **shall**, during `PROCESSING` state and if `AGC` is equal to `earlyControl`, implement an *AGC* that sets the *receive gain* at beginning of the *Rx burst*, to a value that is then kept constant for the remainder of the burst.

`EARLY_AGC_DELAY` (see section 4.8) specifies the delay available after *start time* of a *Rx burst* for an `earlyControl` AGC to have set the *receive gain*.

Rx channels **shall**, during `PROCESSING` state and if `AGC` is equal to `permanentControl`, implement an *AGC* that remains active during the full *Rx burst*.

Further aspects of the implemented *AGC* are *unspecified.*

For *Rx channels* implementing *AGC*, changes in *receive gain* can be indicated to the *radio application* using the `GainChanges` service (see section 2.4.6).

For *Rx channels* implementing a permanent *AGC*, the *AGC* can be deactivated and reactivated using the `AGCActivation` service (see section 2.4.6).

Channelization requirements

Channels **shall** exhibit, during PROCESSING state and so long as *input signal level* is within the *valid input level range*, an actual *transfer function* that fits into the mask defined by fields of CHANNEL_MASK (see section 4.10).

Channels **shall** exhibit, during PROCESSING state and so long as the *input signal* level is within the *valid input level range*, an actual *baseband sampling frequency* (F_s^{BB}) that belongs to CHANNEL_MASK.basebandSamplingFreq ± SAMPLING_FREQ_ACC (see section 4.10).

Channels **shall** exhibit, during PROCESSING state and so long as the *input signal level* is within the *valid input level range*, an actual *carrier frequency* that belongs to applicableCarrierFreq ± CARRIER_FREQ_ACC (see section 4.10).

Termination requirements

The *last processed sample* of a burst **is defined as** the sample of the *processed block* with a *sample number* equal to applicableBurstLength.

Note: value of applicableBurstLength can be set by a *creation operation* (see section 2.4.2) or updated by *setBlockLength()* or *stopBurst()* (see section 3.1.7).

Tx channels **shall**, during PROCESSING state:

- Make the sample of *Tx forwarded block* with *sample number* equal to applicableBurstLength the last sample of the *Tx processed block*,
- Discard any sample of the *Tx forwarded block* after the last sample,
- Use null flushing baseband samples until ramp-down is completed.

Channels **shall** trigger a ProcessingStop transition once *ramp-down* is completed and, for *Tx channels*, once the *Tx forwarded block* has been ended by the *radio application*.

Rx channels **shall**, during PROCESSING state, terminate *down-conversion* without transferring *ramp-down samples* to the *radio application*.

2.3.1.1.6 RESETTING

RESETTING **is specified as** the sub-state of CONFIGURED during which *channels* reset.

RESETTING is completed by *channels* of a *transceiver instance* once:

- Attributes are set back to their *initial values* (see section 3.3),
- Any previously used storage is cleared: for *creation operation* (see sections 3.1.3, 3.1.4, 3.1.5 and 3.1.6), *tuning parameters set* (see section 3.1.11) or *baseband samples* of *Tx channels* (see section 3.1.9).

2.3.1.2 Transitions

2.3.1.2.1 ResetCompleted

ResetCompleted **is specified as** the transition from RESETTING to IDLE.

It is triggered once *channels* have completed the RESETTING state.

2.3.1.2.2 TuningStart

TuningStart **is specified as** the transition from IDLE state to TUNING.

It is triggered under control of CreationControl (see section 2.3.2).

2.3.1.2.3 TuningStop

TuningStop **is specified as** the transition from TUNING to IDLE.

It is triggered once *channels* have completed the TUNING state.

2.3.1.2.4 ProcessingStart

ProcessingStart **is specified as** the transition from IDLE to PROCESSING.

It is triggered under control of CreationControl (see section 2.3.2).

2.3.1.2.5 ProcessingStop

ProcessingStop **is specified as** the transition from PROCESSING to IDLE.

It is triggered by PROCESSING based on knowledge of *last processed sample* (see section 2.3.1.1.5).

2.3.1.2.6 RuntimeReset

RuntimeReset **is specified as** the transition from OPERATING to RESETTING.

It is triggered upon call of *reset()* (see section 3.1.1) by the *radio application*.

2.3.2 CreationControl

CreationControl **is specified as** the autonomous process followed by a *transceiver instance* for the control of creation of the bursts executed by *channels*.

An instance of CreationControl applies to all *Tx channels* of a *transceiver instance*.

An instance of CreationControl applies to all *Rx channels* of a *transceiver instance*.

The following figure is the statechart of CreationControl state machine:

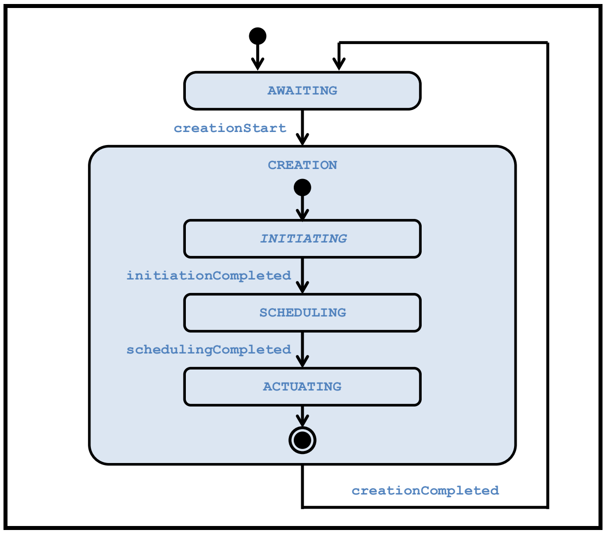

Figure 10 CreationControl statechart

2.3.2.1 States

2.3.2.1.1 AWAITING

AWAITING **is specified as** the state of CreationControl during which a *transceiver instance* stays until it triggers a *burst creation*.

A *transceiver instance* **shall**, during AWAITING, wait until a *creation command* is available in storage.

2.3.2.1.2 CREATING

CREATING **is specified as** the state of CreationControl during which a *transceiver instance* performs creation of a particular burst.

The *burst under creation* **is defined as** the burst that an instance of CREATING aims to create.

The *applied creation command* **is defined as** the *creation command* applied by CREATING for creation of the *burst under creation*.

The entry transition of CREATING **is specified as** a transition to the INITIATING sub-state.

The exit transition of CREATING **is specified as** a transition after completion of the ACTUATION sub-state.

2.3.2.1.3 INITIATING

INITIATING **is specified as** the sub-state of CREATING during which *burst creation* is initiated.

A *transceiver instance* **shall**, during INITIATING, make the oldest *creation command* available in storage the *applied creation command*, and remove it from storage.

A *transceiver instance* **shall**, during INITIATING, set value of applicableBurstLength to value of *requestedLength* as specified in the *applied creation command*.

A *transceiver instance* **shall**, during INITIATING, increment burstCount (see section 3.3.1.1) by 1 (one), rolling-over to 1 after 4.294.967.295 (2^32-1).

A *transceiver instance* **shall**, for *Rx channels* during INITIATING, set the length of *Rx packets* to the value of applicableRxPacketsLength (see section 3.3.1.2).

A *transceiver instance* **shall**, during INITIATING, search for stored *tuning parameters set* according to a condition specified by value of TUNING_ASSOCIATION (see section 4.3):

- For sequential: search for the oldest stored *tuning parameters set*,
- For burstReferencing: search for a stored *tuning parameters set* with value of *requestedBurstNumber* equal to value of burstCount.

A *transceiver instance* **shall**, during INITIATING, if a stored *tuning parameters set* was found, use it as the *applicable tuning parameters set* and remove it from storage.

A *transceiver instance* **shall**, during INITIATING, if no stored *tuning parameters set* was found, set the *applicable tuning parameters set* as follows:

- *requestedTuningPreset* equals to UndefinedTuningPreset,
- *requestedCarrierFreq* equals to UndefinedCarrierFreq,
- *requestedDelay* equals to UndefinedDelay.

2.3.2.1.4 SCHEDULING

SCHEDULING **is specified as** the sub-state of CREATING during which the *start time*, *activation time* and *tuning time* of a *burst under creation* are determined.

Start time corresponds to start of the core burst at radio signal level (see section 1.2.3).

A channel **shall** stay in SCHEDULING until all information required for determination of *start time* is known.

A *channel* **shall**, during SCHEDULING of a *startBurst() creation command*, make *start time* equal to the *termination time* of the previous burst plus INTER-PROCESSING (see section 3.1.4).

A *channel* **shall**, during SCHEDULING of a *scheduleRelativeBurst() creation command*, make *start time* equal to the *start time* of the previous burst on *channels* specified by value of *requestedAlternate* plus the value of *requestedDelay* (see section 3.1.4).

A *channel* **shall**, during SCHEDULING of a *scheduleAbsoluteBurst() creation command*, make *start time* equal to the value of *requestedStartTime* (see section 3.1.5).

A *channel* **shall**, during SCHEDULING of a *scheduleStrobedBurst() creation command*, make *start time* equal to the occurrence time of the next strobe triggered on the *strobe source* specified by *requestedStrobeSource* plus the value of *requestedDelay* (see section 3.1.6).

Activation time **is defined as** the time at which the startProcessing transition is triggered.

A *channel* **shall**, during SCHEDULING, determine *activation time* so that the effective *start time* belongs to *start time* ± START_TIME_ACC (see section 4.12).

Note: for *Tx channels*, *activation time* is equal to *start time* minus *up-conversion latency* (see Figure 2); for *Rx channels*, *activation time* is equal to *start time* (see Figure 5).

Tuning time **is defined as** the time at which the `startTuning` transition is triggered to ensure that the *applicable tuning parameters set* is implemented by the `TUNING` state with a `TuningStop` transition triggered before *activation time*.

A *channel* **shall**, during `SCHEDULING`, determine *tuning time* based on *activation time*.

2.3.2.1.5 ACTUATING

`ACTUATING` **is specified as** the sub-state of `CREATING` during which the *transceiver instance* triggers `TuningStart` and `ProcessingStart` transitions of the `Channels` state machine.

A *transceiver instance* **shall**, during `ACTUATING`, trigger a `TuningStart` transition at *tuning time.*

A *transceiver instance* **shall**, during `ACTUATING` of *Tx channels* if the *applied creation operation* is *startBurst()*, shift *activation time* until *first baseband sample* becomes available.

A *transceiver instance* **shall**, during `ACTUATING`, trigger a `ProcessingStart` transition at *activation time*.

2.3.2.2 Transitions

2.3.2.2.1 CreationStart

`CreationStart` **is specified as** the transition from `AWAITING` to `CREATION`.

It is triggered once a *creation command* is available in storage.

2.3.2.2.2 InitiationCompleted

`CreationStart` **is specified as** the transition from `INITIATING` to `SCHEDULING`.

It is triggered once a *transceiver instance* has completed `INITIATING`.

2.3.2.2.3 SchedulingCompleted

`SchedulingCompleted` **is specified as** the transition from `SCHEDULING` to `ACTUATING`.

It is triggered once a *transceiver instance* has completed `SCHEDULING`.

2.3.2.2.4 CreationCompleted

`CreationCompleted` **is specified as** the transition from `CREATION` to `AWAITING`.

It is triggered once a *transceiver instance* has completed `ACTUATING`.

2.3.3 RadioSilence

`RadioSilence` **is specified as** the state machine applicable if `RadioSilence` is active or if the channels can be turned to radio silence by an agent different from the *radio application*.

The following figure is the statechart of RadioSilence:

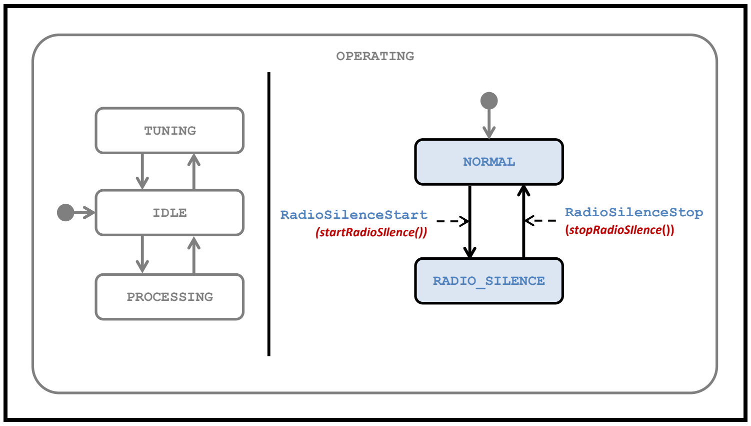

Figure 11 RadioSilence statechart

RadioSilence is a sub-state machine of OPERATING, parallel to the sub-state machine of OPERATING specified by Channels (see section 2.3.1).

2.3.3.1 States

2.3.3.1.1 NORMAL

NORMAL **is specified as** the state during which the *channels* operate as specified for the OPERATING state of Channels.

The entry transition of RadioSilence brings to NORMAL.

2.3.3.1.2 RADIO_SILENCE

RADIO_SILENCE **is specified as** the state during which *channels* minimize the radiated radio signal, preventing respect of the specified *tuning* during PROCESSING state.

The RADIO_SILENCE state does not impact any other aspect of the OPERATING state.

2.3.3.2 Transitions

2.3.3.2.1 RadioSilenceStart

RadioSilenceStart **is specified as** the transition from NORMAL to RADIO_SILENCE.

It is triggered by invocation of *startRadioSilence()*.

2.3.3.2.2 RadioSilenceStop

RadioSilenceStop **is specified as** the transition from RADIO_SILENCE to NORMAL.

It is triggered by invocation of *stopRadioSilence()*.

2.3.4 Retuning

Retuning **is specified as** the state machine applicable if Retuning is active.

The following figure is the statechart of Retuning:

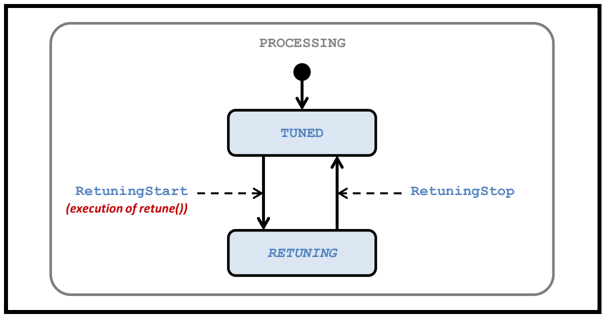

Figure 12 Retuning statechart

Retuning is a sub-state machine of PROCESSING (see section 2.3.1).

2.3.4.1 States

2.3.4.1.1 TUNED

TUNED **is specified as** the sub-state of PROCESSING during which *channels* process with stable tuning characteristics that comply with the specified *tuning*.

The entry transition of Retuning brings to TUNED.

2.3.4.1.2 RETUNING

RETUNING **is specified as** the sub-state of PROCESSING during which *channels* change their *tuning* while continuing to perform *up-conversion* or *down-conversion*.

RETUNING_DURATION (see section 4.8) specifies the maximum duration of RETUNING state.

2.3.4.2 Transitions

2.3.4.2.1 RetuningStart

RetuningStart **is specified as** the transition from TUNED to RETUNING.

It is triggered when the *radio application* calls *retune()* (see section 3.1.12).

2.3.4.2.2 RetuningStop

RetuningStop **is specified as** the transition from RETUNING to TUNED.

It is triggered when the new tuning characteristics are stable and conform to the tuning changes commanded by *retune()*.

2.4 Services groups description

The class diagrams appearing in this section aim to comply with the OMG Unified Modeling Language v2.5, as specified in [Ref4].

2.4.1 Transceiver::Management

The **Management** services group enables *radio applications* to manage the *Transceiver*, and contains the following services:

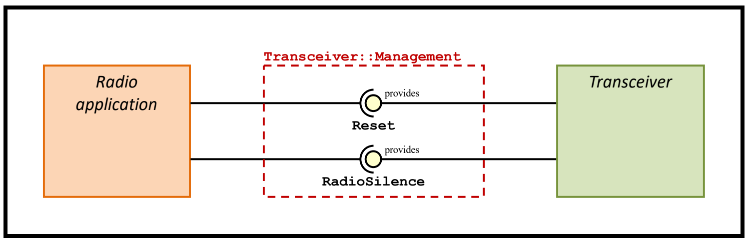

Figure 13 Services of Management services group

The **Reset** service enables *radio applications* to reset *channels*.

The **RadioSilence** service enables *radio applications* to start and stop *radio silence*.

2.4.1.1 Transceiver::Management::Reset Interface Description

The **Reset** interface is composed of the *reset()* operation, as depicted in the following figure:

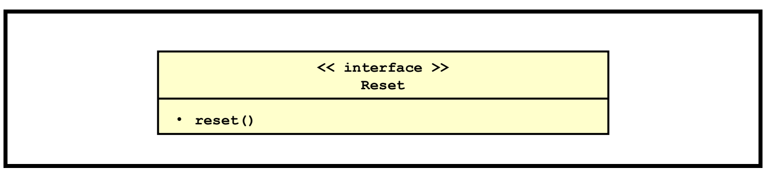

Figure 14 Management::Reset interface

reset() enables *radio applications* to reset *channels*.

2.4.1.2 Transceiver::Management::RadioSilence Interface Description

The **RadioSilence** interface is composed of the *startRadioSilence()* and *stopRadioSilence()* operations, as depicted in the following figure:

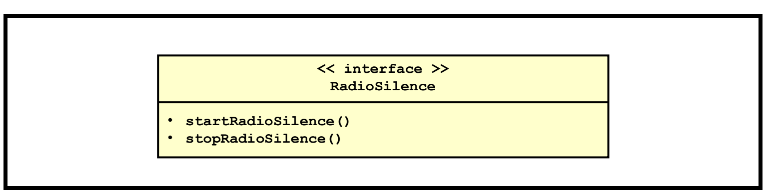

Figure 15 Management::RadioSilence interface

startRadioSilence() enables *radio applications* to start *radio silence*.

stopRadioSilence() enables *radio applications* to stop *radio silence*.

2.4.2 Transceiver::BurstControl

The **BurstControl** services group enables *radio applications* to control the creation of *bursts*, and contains the following services:

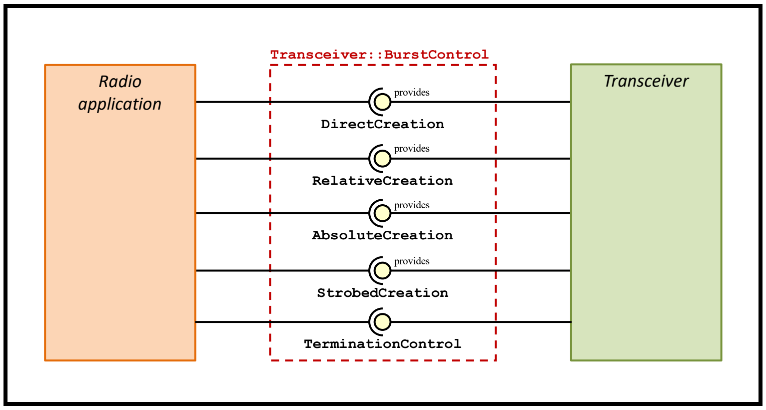

Figure 16 Services of BurstControl services group

A *creation service* **is defined as** a *service* of **BurstControl** services group.

A *creation operation* **is defined as** one *operation* of a *creation service*: *startBurst()*, *scheduleRelativeBurst()*, *scheduleAbsoluteBurst()* or *scheduleStrobedBurst()*.

The **DirectCreation** service enables *radio applications* to schedule the creation of a new burst with no specific requirement on its *start time*.

A *timely creation service* **is defined as** a **RelativeCreation**, **AbsoluteCreation** or **StrobedCreation** *service*.

A *timely creation operation* **is defined as** a *creation operation* of a *timely creation service*: *scheduleRelativeBurst()*, *scheduleAbsoluteBurst()* or *scheduleStrobedBurst()*.

Timely creation services and *operations* enables to specify the *start time* of scheduled burst, measured at the *radio signal* level, as specified in section 1.2.

The **RelativeCreation** service enables *radio applications* to schedule the creation of a new burst with a *start time* delayed by a specified value from the *start time* of the previous *burst*.

The **AbsoluteCreation** service enables *radio applications* to schedule the creation of a new burst with a *start time* specified using the *transceiver time*.

The **StrobedCreation** service enables *radio applications* to schedule the creation of a new burst with a *start time* delayed by a specified value from the next occurrence of a strobe discrete signal on a specified strobe source.

All *creation services* enable *radio applications* to specify the length of the *baseband block*.

Radio applications must make calls to *creation operations* in the same order as the order of created bursts (see section 2.3.2), and can make up to **CREATION_STORAGE** (see section 4.8) anticipated calls to *creation operations*.

Radio applications must make calls to *timely creation operations* ensuring value of **INTER-PROCESSING** (see section 4.8) is respected.

The **Termination** service enables *radio applications* to control termination of an ongoing *processing phase*.

2.4.2.1 Transceiver::BurstControl::DirectCreation Interface Description

The **DirectCreation** interface is composed of the *startBurst()* operation, as depicted in the following figure:

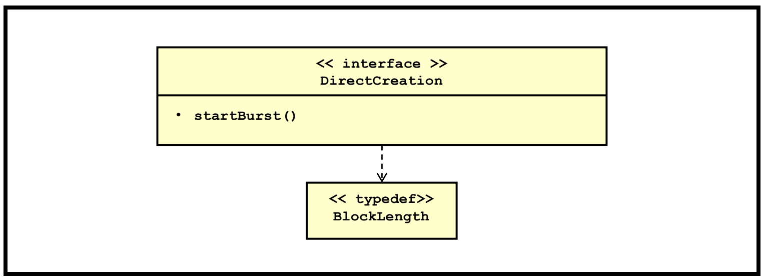

Figure 17 BurstControl::DirectCreation interface

startBurst() enables *radio applications* to schedule the creation of a new burst with no specific requirement on its *start time*.

2.4.2.2 Transceiver::BurstControl::RelativeCreation Interface Description

The **RelativeCreation** interface is composed of the *scheduleRelativeBurst()* operation, as depicted in the following figure:

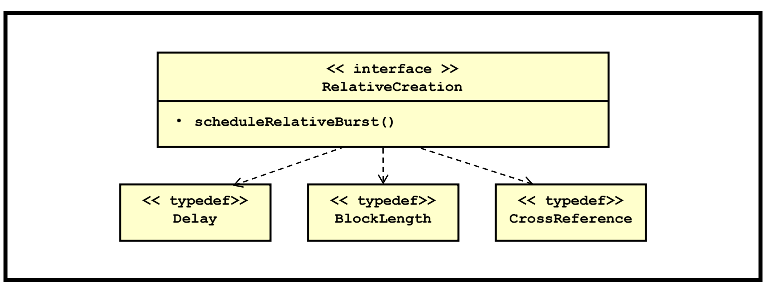

Figure 18 BurstControl::RelativeCreation interface

scheduleRelativeBurst() enables *radio applications* to schedule the creation of a new burst with a *start time* delayed by a specified value from the *start time* of the previous *burst*.

scheduleRelativeBurst() must be combined with another creation operation (e.g. *startBurst()* or *scheduleStrobedBurst()*), used to create the first *burst* of all series of *bursts* then created using *scheduleRelativeBurst()*.

2.4.2.3 Transceiver::BurstControl::AbsoluteCreation Interface Description

The **AbsoluteCreation** interface is composed of the *scheduleAbsoluteBurst()* operation, as depicted in the following figure:

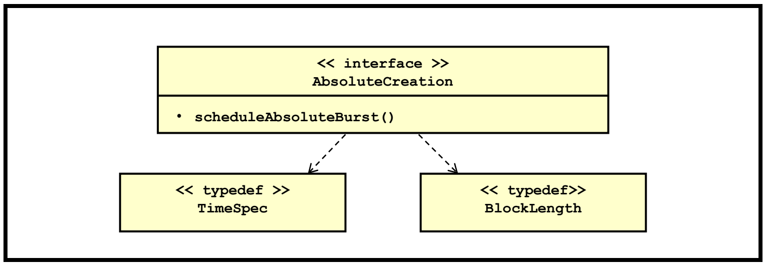

Figure 19 BurstControl::AbsoluteCreation interface

scheduleAbsoluteBurst() enables *radio applications* to schedule the creation of a new burst with a *start time* specified using the *transceiver time*.

scheduleAbsoluteBurst() must be used in conjunction with a mechanism enabling *radio applications* to get the *transceiver time* (e.g. the **TransceiverTime** service).

2.4.2.4 Transceiver::BurstControl::StrobedCreation Interface Description

The **StrobedCreation** interface is composed of the *scheduleStrobedBurst()* operation, as depicted in the following figure:

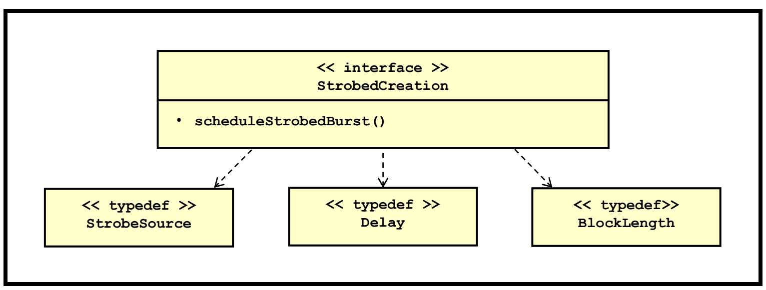

Figure 20 BurstControl::StrobedCreation interface

scheduleStrobedBurst() enables *radio applications* to schedule the creation of a new burst with a *start time* delayed by a specified value from the next occurrence of a strobe discrete signal on a specified strobe source.

The specified strobe source can be internal to the platform (e.g. the PPS signal of a GNSS system) or be provided by a component of the *radio application* (e.g. a FPGA component).

2.4.2.5 Transceiver::BurstControl::Termination Interface Description

The **Termination** interface is composed of the *setBlockLength()* and *stopBurst()* operations, as depicted in the following figure:

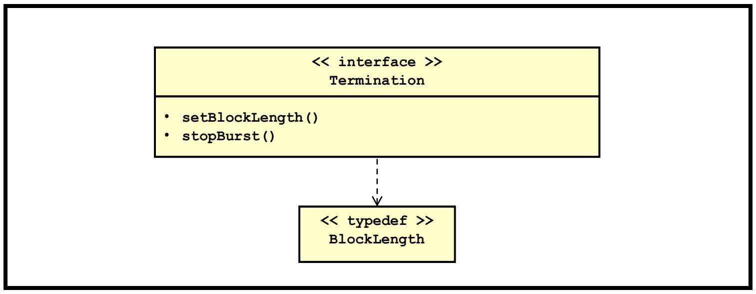

Figure 21 BurstControl::Termination interface

setBlockLength() enables *radio applications* to set the length of the *baseband block* processed by *channels* during an ongoing *processing phase*.

stopBurst() enables *radio applications* command immediate termination of an ongoing *processing phase*.

2.4.3 Transceiver::BasebandSignal

The **BasebandSignal** services group enables *radio applications* to exchange blocks of baseband samples processed by *channel*, and contains the following services:

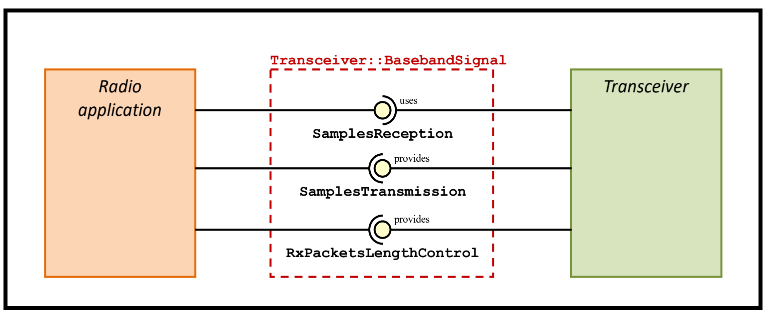

Figure 22 Services of BasebandSignal services group

The **SamplesReception** service enables *radio applications* to obtain a *receive baseband block* from an *Rx channel* during a *reception* (see section 1.2.4).

The **SamplesTransmission** service enables *radio applications* to forward a *transmit baseband block* to a *Tx channel* during a *transmission* (see section 1.2.5).

The **RxPacketsLengthControl** service enables *radio applications* to set the value of the **applicableRxPacketsLength** attribute.

2.4.3.1 Transceiver::BasebandSignal::SamplesReception Interface Description

The **SamplesReception** interface is composed of the *pushRxPacket()* operation, as depicted in the following figure:

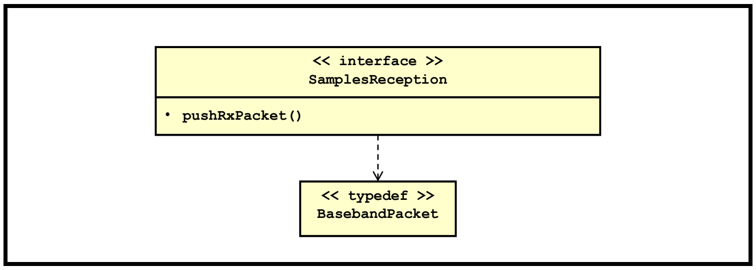

Figure 23 BasebandSignal::SamplesReception interface

pushRxPacket() enables *radio applications* to obtain a *baseband packet* from an *Rx channel* and to be specified if the packet is the *last packet* of the *Rx block*.

2.4.3.2 Transceiver::BasebandSignal::SamplesTransmission Interface Description

The **SamplesTransmission** interface is composed of the *pushTxPacket()* operation, as depicted in the following figure:

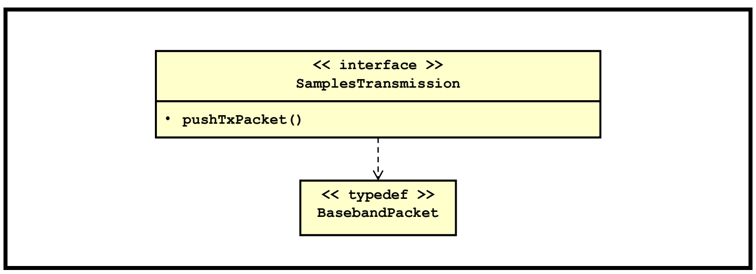

Figure 24 BasebandSignal::SamplesTransmission interface

pushTxPacket() enables *radio applications* to forward a *baseband packet* to a *Tx channel* and to specify if the packet is the *last packet* of the *Tx forwarded block*.

2.4.3.3 Transceiver::BasebandSignal::RxPacketsLengthControl Interface Description

The **RxPacketsLengthControl** interface is composed of the *setRxPacketsLength()* operation, as depicted in the following figure:

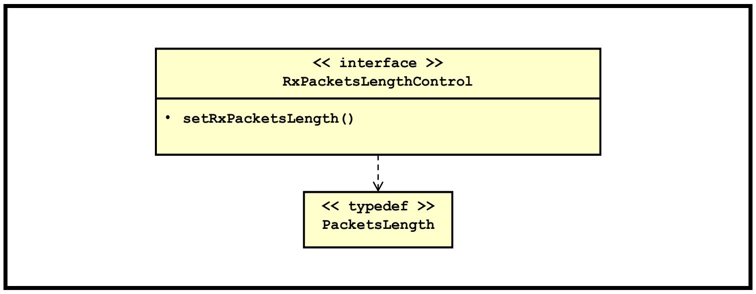

Figure 25 BasebandSignal::RxPacketsLengthControl interface

setRxPacketsLength() enables *radio applications* to set the value of the **applicableRxPacketsLength** attribute.

2.4.4 Transceiver::Tuning

The **Tuning** services group enables *radio applications* to control the tuning of *bursts*, and contains the following services:

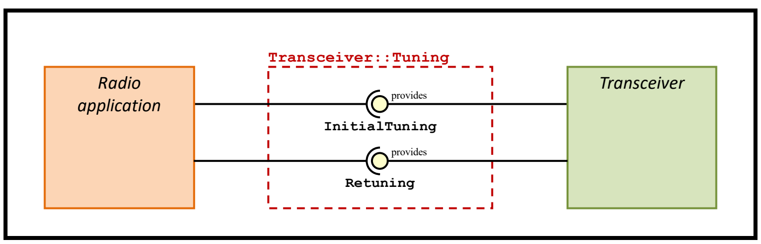

Figure 26 Services of Tuning services group

The **InitialTuning** service enables *radio applications* to specify the *tuning preset*, *carrier frequency* and *gain* values to be applied at beginning of a future *burst*.

Radio applications can make up to **TUNING_STORAGE** (see section 4.8) anticipated calls to *setTuning()*.

Radio applications must use the **InitialTuning** service for a given *burst*, if needed, before the stored creation operation of the *burst* is used by **CreationControl** (see section 2.3.2).

The **Retuning** service enables *radio applications* to schedule and specify new values of *carrier frequency* and *gain* without interrupting an ongoing *processing phase*.

2.4.4.1 Transceiver::Tuning::InitialTuning Interface Description

The **InitialTuning** interface is composed of the *setTuning()* operation, as depicted in the following figure:

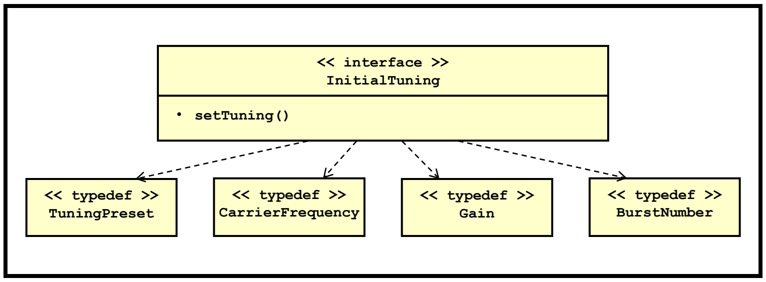

Figure 27 Tuning::InitialTuning interface

setTuning() enables *radio applications* to specify the *tuning preset*, *carrier frequency* and *gain* values to be applied at beginning of a future *burst*.

2.4.4.2 Transceiver::Tuning::Retuning Interface Description

The **Retuning** interface is composed of *retune()* operation, as depicted in the following figure:

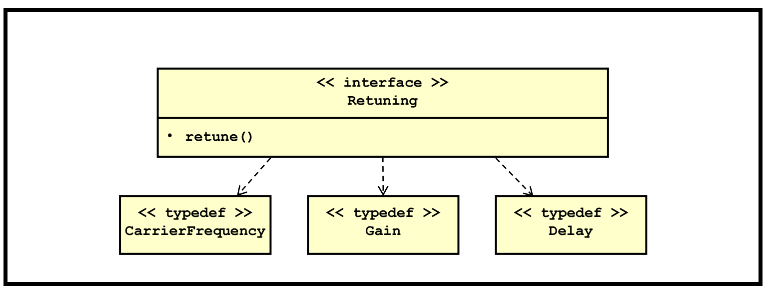

Figure 28 Tuning::Retuning interface

retune() enables *radio applications* to schedule and specify new values of *carrier frequency* and *gain* without interrupting an ongoing *processing phase*.

2.4.5 Transceiver::Notifications

The **Notifications** services group enables *radio applications* to be notified by *channels* of execution events and execution errors, and contains the following services:

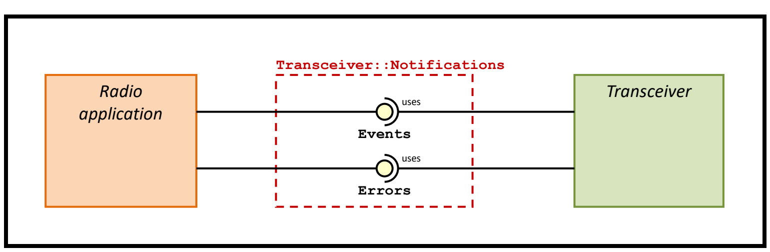

Figure 29 Services of Notifications services group

The **Events** service enables *radio applications* to be notified of *events* occurrences.

The **Errors** service enables *radio applications* to be notified of *errors* occurrences.

2.4.5.1 Transceiver::Notifications::Events Interface Description

The **Events** interface is composed of the *notifyEvent()* operation, as depicted in the following figure:

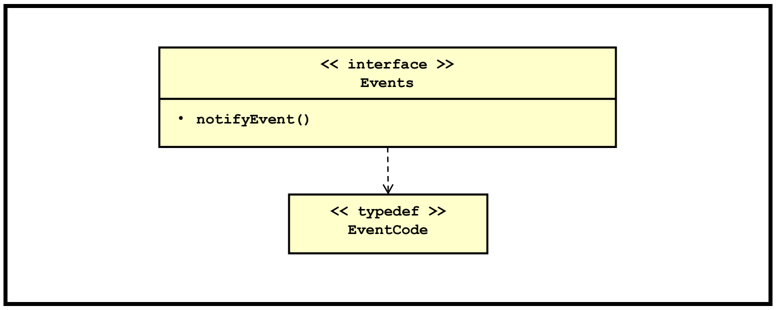

Figure 30 Notifications::Events interface

notifyEvent() enables *radio applications* to be notified of *events* occurrences.

2.4.5.2 Transceiver::Notifications::Errors Interface Description

The **Errors** interface is composed of the *notifyError()* operation, as depicted in the following figure:

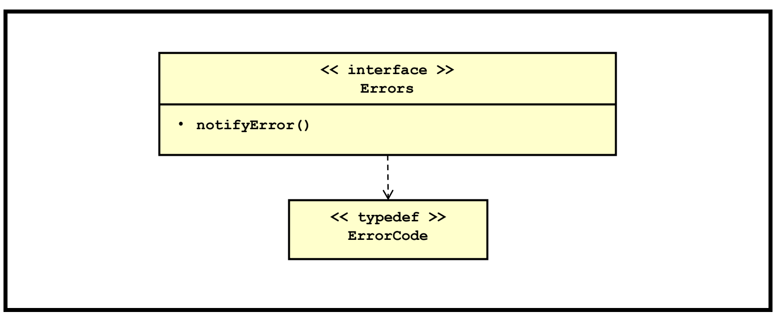

Figure 31 Notifications::Errors interface

notifyError() enables *radio applications* to be notified of *errors* occurrences.

2.4.6 Transceiver::GainControl

The **GainControl** services group enables *radio applications* to be informed of aspects related to gain control, and contains the following service:

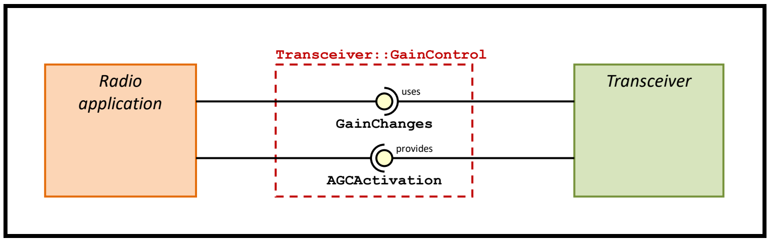

Figure 32 Services of GainControl services group

The **GainCanges** service enables *radio applications* to be notified of changes in *gain* values decided by *channels* during a *processing phase*.

The **AGCActivation** service enables *radio applications* to deactivate and reactivate permanent *AGC* while a *reception* is ongoing.

2.4.6.1 Transceiver::GainControl::GainChanges Interface Description

The **GainChanges** interface is composed of the *indicateGain()* operation, as depicted in the following figure:

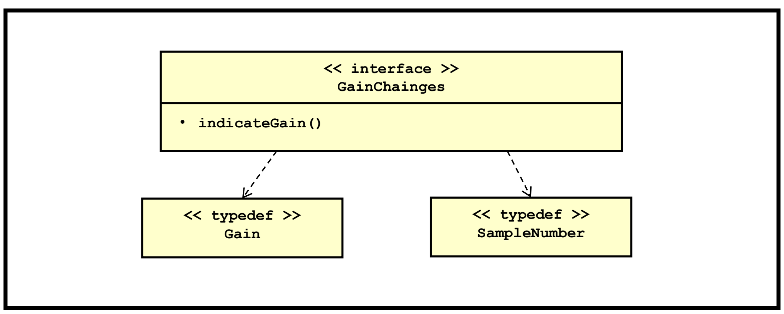

Figure 33 GainControl::GainChanges interface

indicateGain() enables *radio applications* to be notified of changes in *gain* values decided by *channels* during a *processing phase*.

2.4.6.2 Transceiver::GainControl::AGCActivation Interface Description

The **AGCActivation** interface is composed of the *deactivateAGC()* and *reactivateAGC()* operations, as depicted in the following figure:

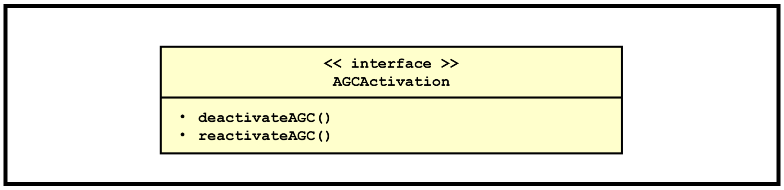

Figure 34 GainControl::AGCActivation interface

deactivateAGC() enables *radio applications* to deactivate *AGC* in the course of a *reception*.

reactivateAGC() enables *radio applications* to reactivate a previously deactivated *AGC*.

2.4.7 Transceiver::TransceiverTime

The **TransceiverTime** services group enables *radio applications* to get values of *transceiver time*, and contains the following service:

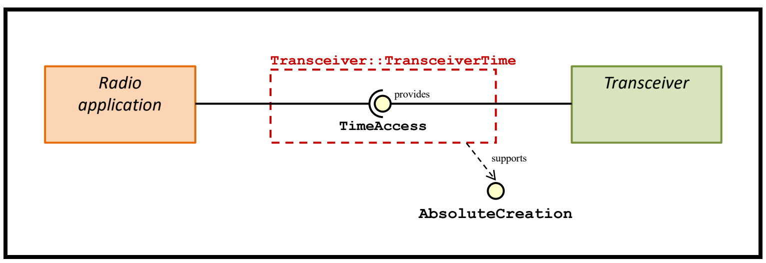

Figure 35 Service of TransceiverTime services group

The **TimeAccess** service enables *radio applications* to get the current value of *transceiver time* and the value of *transceiver time* for the *start time* of the last created burst.

2.4.7.1 Transceiver::TransceiverTime::TimeAccess Interface Description

The **TimeAccess** interface is composed of the *getCurrentTime()* and *getLastStartTime()* operations, as depicted in the following figure:

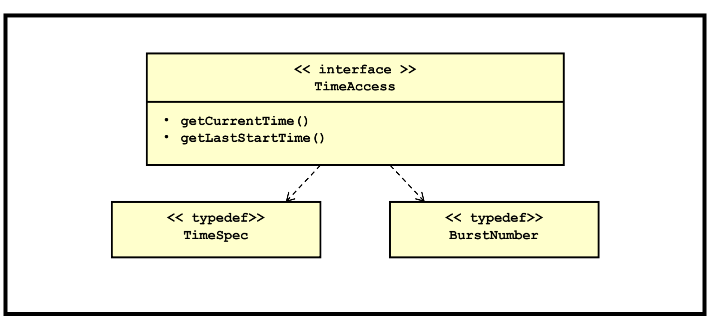

Figure 36 TransceiverTime::TimeAccess interface

getCurrentTime() enables *radio applications* to get the current value of *transceiver time*.

getLastStartTime() enables *radio applications* to get the value of *transceiver time* for the *start time* of the last created burst.

2.4.8 Transceiver::Strobing

The **Strobing** services group enables *radio applications* to trigger strobes that can be used for creation of bursts scheduled with **StrobedCreation** service, and contains the following interface:

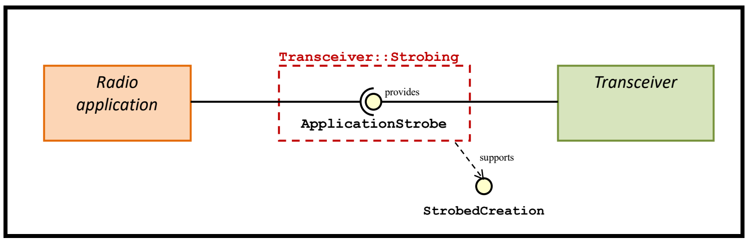

Figure 37 Service of Strobing services group

The **Strobing** service enables *radio applications* to trigger strobes that can be used for creation of a bursts scheduled with **StrobedCreation** service.

2.4.8.1 Transceiver::Strobing::ApplicationStrobe Interface Description

The **ApplicationStrobe** interface is composed of the *triggerStrobe()* operation, as depicted in the following figure:

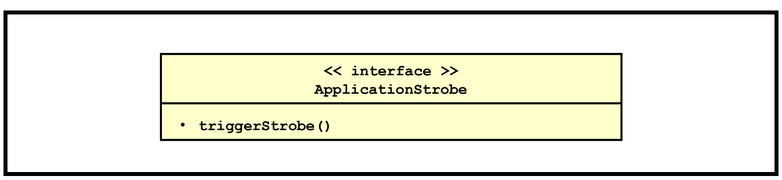

Figure 38 Strobing::ApplicationStrobe interface

triggerStrobe() enables *radio applications* to trigger occurrences of strobes that can be used for creation of a burst scheduled by a *scheduleStrobedBurst()* call (see section 3.1.6) with *requestedStrobeSource* parameter equal to **ApplicationStrobe**.

3 Service primitives and attributes

3.1 Service primitives

This section specifies the *primitives* of the *services interfaces* of the *Transceiver API*.

Each declaration of a *primitive* complies with the Full PIM IDL Profile of WInnForum *IDL profiles for PIM of SDR Applications,* specified in [Ref5].

The conformance criteria for Application-Specific Interfaces is applied (see [Ref5], section 1.3.2): "*An Application-Specific Interface is conformant with one applicable IDL Profile if each of its operations exclusively uses capabilities of the applicable IDL Profile.*".

The declaration of each primitive also complies with SCA 4.1 Appendix E-1 [Ref6].

The specified declarations are common normative inputs for the *PSMs* (see section 1.1) specified in appendices of the *specification*.

The sequence diagrams appearing in this section are based on the OMG Unified Modeling Language v2.5, as specified in [Ref4].

3.1.1 Transceiver::Management::Reset

3.1.1.1 reset Operation

3.1.1.1.1 Overview

reset() commands *channels* to reset.

3.1.1.1.2 Associated properties

RESET_WCET (see section 4.15) specifies the worst-case execution time of the primitive.

3.1.1.1.3 Declaration

The declaration of the operation **is specified as**:

```
void reset();
```

3.1.1.1.4 Parameters

None.

3.1.1.1.5 Returned value

None.

3.1.1.1.6 Originator

Radio application.

3.1.1.1.7 Exceptions

None.

3.1.1.1.8 Behavior requirements

An *active instance* of **Reset** **shall**, on a call to *reset()*:

- Stop any ongoing activity,
- Trigger a **RuntimeReset** transition (see section 2.3.1.2.6),
- Complete the **RESETTING** state (see section 2.3.1.1.6),
- Trigger a **ResetCompleted** transition (see section 2.3.1.2.1),
- Return the call to the *radio application*.

3.1.2 Transceiver::Management::RadioSilence

3.1.2.1 startRadioSilence Operation

3.1.2.1.1 Overview

startRadioSilence() commands *Tx channels* to start a *radio silence* phase, as depicted in the following figure:

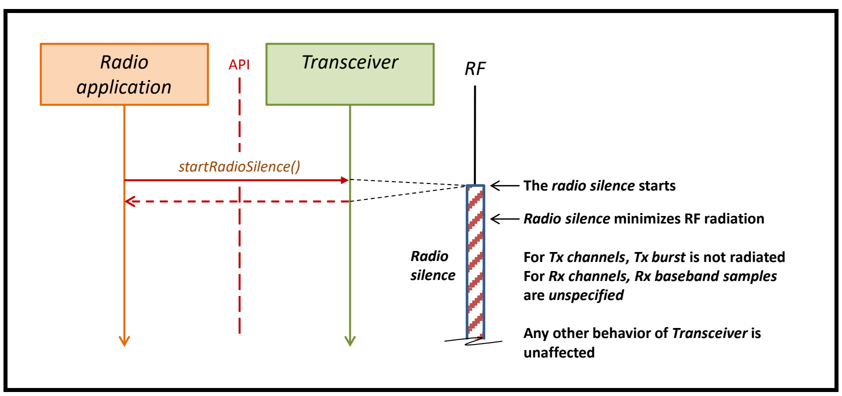

Figure 39 Principle of *startRadioSilence()*

3.1.2.1.2 Associated properties

START_SILENCE_WCET (see section 4.15) specifies the worst-case execution time of the primitive.

3.1.2.1.3 Declaration

The declaration of the operation **is specified as**:

```
void startRadioSilence();
```

3.1.2.1.4 Parameters

None.

3.1.2.1.5 Returned value

None.

3.1.2.1.6 Originator

Radio application.

3.1.2.1.7 Exceptions

None.

3.1.2.1.8 Behavior requirements

An *active instance* of **RadioSilence** **shall**, on a call to *startRadioSilence()*:

- Trigger a **RadioSilenceStart** transition (see section 2.3.2),
- Stop radiating any signal at RF level,
- Return the call to the *radio application*.

The **RADIO_SILENCE** state does not impact operation of the **OPERATING** state further than preventing RF radiation.

3.1.2.2 stopRadioSilence Operation

3.1.2.2.1 Overview

stopRadioSilence() commands the *Tx channels* to stop a radio silence phase, as depicted in the following figure:

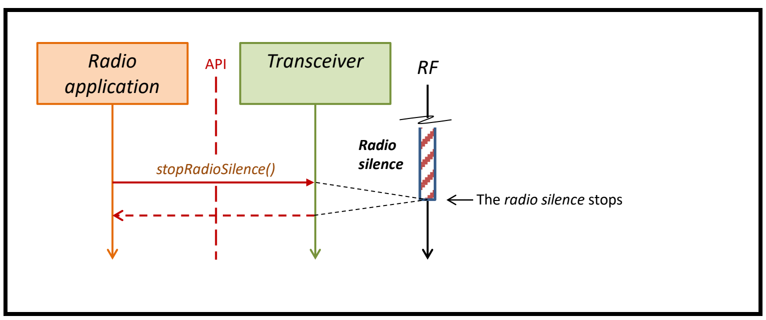

Figure 40 Principle of *stopRadioSilence()*

3.1.2.2.2 Associated properties

STOP_SILENCE_WCET (see section 4.15) specifies the worst-case execution time of the primitive.

3.1.2.2.3 Declaration

The declaration of the operation **is specified as**:

```
void stopRadioSilence();
```

3.1.2.2.4 Parameters

None.

3.1.2.2.5 Returned value

None.

3.1.2.2.6 Originator

Radio application.

3.1.2.2.7 Exceptions

None.

3.1.2.2.8 Behavior requirements

An *active instance* of **RadioSilence** **shall**, on a call to *stopRadioSilence()*:

- Trigger a **RadioSilenceStop** transition (see section 2.3.2),
- Resume normal radio operation at RF level,
- Return the call to the *radio application*.

3.1.3 Transceiver::BurstControl::DirectCreation

3.1.3.1 startBurst Operation

3.1.3.1.1 Overview

startBurst() commands the *channels* to schedule creation of a burst with no specified *start time*, as depicted in the following figure:

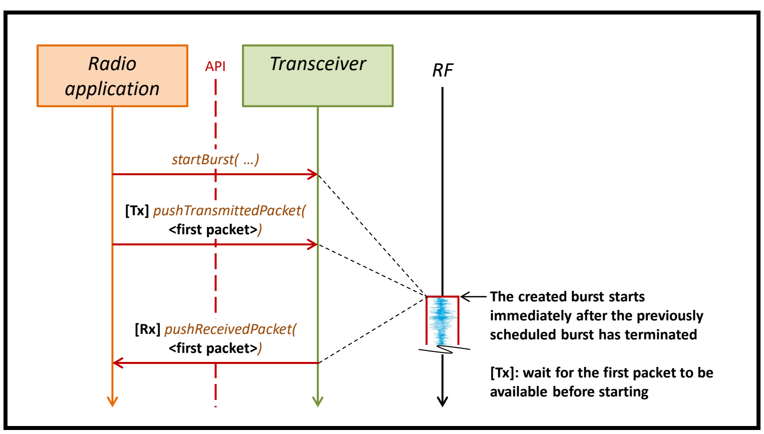

Figure 41 Principle of *startBurst()*

3.1.3.1.2 Associated properties

CREATION_STORAGE (see section 4.8) specifies the maximum number of calls to *creation operations*, such as calls to *startBurst()*, which *channels* can store in advance.

DIRECT_WCET (see section 4.15) specifies the worst-case execution time of the primitive.

3.1.3.1.3 Declaration

The declaration of the operation **is specified as**:

```
void startBurst(
    in BlockLength requestedLength);
```

3.1.3.1.4 Parameters

Name	Type	Description
requestedLength	***BlockLength*** (see § 3.4.3)	Value of **applicableLength**. Number of *baseband samples* to be processed during the *processing phase* associated to the *burst*: ▪ If equal to **UndefinedBlockLength**: specifies an *undefined* value, ▪ If not equal to **UndefinedBlockLength**: specifies the *number* of *baseband samples* of the *baseband block* to be processed during **PROCESSING** (see section 2.3.1).

Table 3 Specification of *startBurst()* parameters

The *parameters validity properties* **are specified as** (see section 4.7):

- For *requestedLength*: **MIN_BLOCK_LENGTH** and **MAX_BLOCK_LENGTH**.

3.1.3.1.5 Return value

None.

3.1.3.1.6 Originator

Radio application.

3.1.3.1.7 Exceptions

The *exceptions* of the *operation* **are specified as** (see section 3.2):

- **MinBlockLength** and **MaxBlockLength**.

3.1.3.1.8 Behavior requirements

An *active instance* of **DirectCreation** **shall**, on a call to *startBurst()*, handle the *exceptions* of the *operation* as specified in section 3.2.

An *active instance* of **DirectCreation** **shall**, on a call to *startBurst()* that raised no exception:

- If **CREATION_STORAGE** calls (see section 4.8) are stored, wait until storage becomes available,
- Store the call for later usage by **CreationControl** (see section 2.3.2),
- Return the call to the *radio application*.

3.1.4 Transceiver::BurstControl::RelativeCreation

3.1.4.1 scheduleRelativeBurst Operation

3.1.4.1.1 Overview

scheduleRelativeBurst() commands the *channels* to schedule creation of a burst starting at a specified delay after the start time of the previous burst of the referenced channel, as depicted in the following figure:

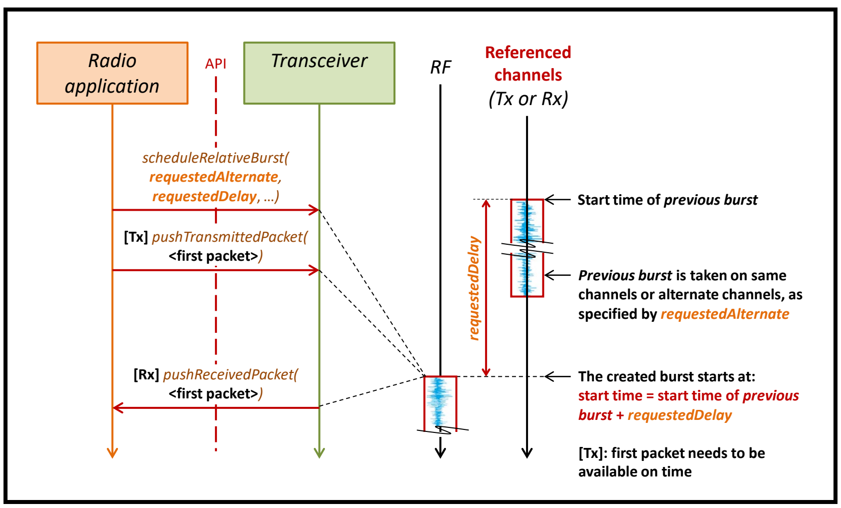

Figure 42 Principle of *scheduleRelativeBurst()*

3.1.4.1.2 Associated properties

CREATION_STORAGE (see section 4.8) specifies the maximum number of calls to *creation operations* that *channels* can store in advance, including calls to *scheduleRelativeBurst()*.

RELATIVE_MILT (see section 4.13) specifies the minimum invocation lead time for correct real-time usage of the operation.

RELATIVE_WCET (see section 4.15) specifies the worst-case execution time of the primitive.

3.1.4.1.3 Declaration

The declaration of the operation **is specified as**:

```
void scheduleRelativeBurst(
    in boolean requestedAlternate,
    in Delay requestedDelay,
    in BlockLength requestedLength);
```

3.1.4.1.4 Parameters

Name	Type	Description
requestedAlternate	***boolean***	For duplex *transceivers*, specifies the *reference channels*: ▪ If equal to **false**: *called channels* are used, ▪ If equal to **true**: *alternate channels* are used.
requestedDelay	***Delay*** (see § 3.4.7).	Specifies the delay between the *start time* of the previous burst scheduled by *reference channel* and the *start time* of the burst to create.
requestedLength	***BlockLength*** (see § 3.4.3)	Number of *baseband samples* to be processed during the *processing phase* associated to the *burst*: ▪ If equal to **UndefinedBlockLength**: specifies an *undefined* value, ▪ If not equal to **UndefinedBlockLength**: specifies the *number* of *baseband samples* of the *baseband block* to be processed during **PROCESSING** (see section 2.3.1).

Table 4 Specification of *scheduleRelativeBurst()* parameters

The *parameters validity properties* **are specified as** (see section 4.7):

- For *requestedAlternate*: **ALTERNATE_REFERENCING**,
- For *requestedDelay*: **MIN_FROM_PREVIOUS** and **MAX_FROM_PREVIOUS**,
- For *requestedLength*: **MIN_BLOCK_LENGTH** and **MAX_BLOCK_LENGTH**.

3.1.4.1.5 Return value

None.

3.1.4.1.6 Originator

Radio application.

3.1.4.1.7 Exceptions

The *exceptions* of the *operation* **are specified as** (see section 3.2):

- **NoAlternateReferencing**,
- **MinFromPrevious** and **MaxFromPrevious**,
- **MinBlockLength** and **MaxBlockLength**,
- **RelativeMILT**.

3.1.4.1.8 Behavior requirements

An *active instance* of **RelativeCreation** **shall**, on a call to *scheduleRelativeBurst()*, handle the *exceptions* of the *operation* as specified in section 3.2.

An *active instance* of **RelativeCreation** **shall**, on a call to *scheduleRelativeBurst()* that raised no exception:

- If **CREATION_STORAGE** calls (see section 4.6) are stored, wait until storage becomes available,
- Store the call for later usage by CreationControl (see section 2.3.2),
- Return the call to the *radio application*.

3.1.5 Transceiver::BurstControl::AbsoluteCreation

3.1.5.1 scheduleAbsoluteBurst Operation

3.1.5.1.1 Overview

scheduleAboluteBurst() commands the *channels* to schedule creation of a burst for which *core burst* will start at the specified *requestedStartTime*, as depicted in the following figure:

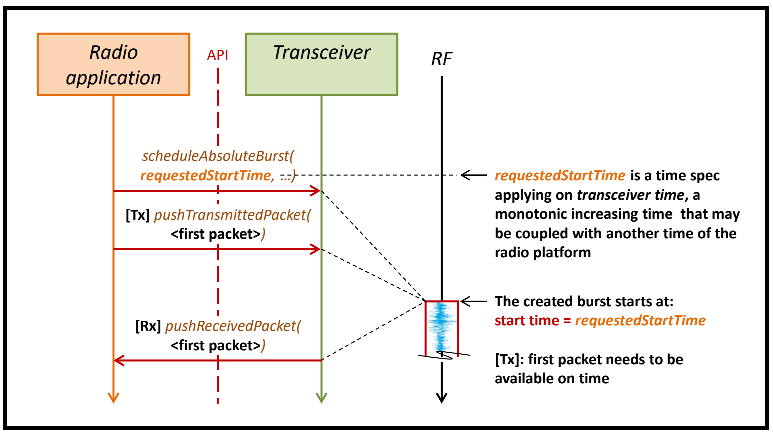

Figure 43 Principle of *scheduleAbsoluteBurst()*

3.1.5.1.2 Associated properties

CREATION_STORAGE (see section 4.8) specifies the maximum number of calls to *creation operations* that *channels* can store in advance, including calls to *scheduleAbsoluteBurst()*.

TIME_COUPLING (see section 4.2) specifies coupling of *transceiver time* with other times of the radio platform.

ABSOLUTE_MILT (see section 4.13) specifies the minimum invocation lead time for correct real-time usage of the operation.

ABSOLUTE_WCET (see section 4.15) specifies the worst-case execution time of the primitive.

3.1.5.1.3 Declaration

The declaration of the operation **is specified as**:

```
void scheduleAbsoluteBurst(
    in TimeSpec requestedStartTime,
    in BlockLength requestedLength);
```

3.1.5.1.4 Parameters

Name	Type	Description
requestedStartTime	***TimeSpec*** (see § 3.4.16)	Specifies the value of *start time* of the burst to create, expressed according to *transceiver time*.
requestedLength	***BlockLength*** (see § 3.4.3)	Number of *baseband samples* to be processed during the *processing phase* associated to the *burst*: ▪ If equal to **UndefinedBlockLength**: specifies an *undefined* value, ▪ If not equal to **UndefinedBlockLength**: specifies the *number* of *baseband samples* of the *baseband block* to be processed during **PROCESSING** (see section 2.3.1).

Table 5 Specification of *scheduleAbsoluteBurst()* parameters

The *parameters validity properties* **are specified as** (see section 4.7):

- For *requestedLength*: **MIN_BLOCK_LENGTH** and **MAX_BLOCK_LENGTH**.

3.1.5.1.5 Return value

None.

3.1.5.1.6 Originator

Radio application.

3.1.5.1.7 Exceptions

The *exceptions* of the *operation* **are specified as** (see section 3.2):

- **MaxNanoseconds**,
- **MinBlockLength** and **MaxBlockLength**,
- **AbsoluteMILT**.

3.1.5.1.8 Behavior requirements

An *active instance* of **AbsoluteCreation** **shall**, on a call to *scheduleAbsoluteBurst()*, handle the *exceptions* of the *operation* as specified in section 3.2.

An *active instance* of **AbsoluteCreation** **shall**, on a call to *scheduleAbsoluteBurst()* that raised no exception:

- If **CREATION_STORAGE** calls (see section 4.8) are stored, wait until storage becomes available,
- Store the call for later usage by **CreationControl** (see section 2.3.2),
- Return the call to the *radio application*.

3.1.6 Transceiver::BurstControl::StrobedCreation

3.1.6.1 scheduleStrobedBurst Operation

3.1.6.1.1 Overview

*scheduleStrobedBurst()*commands the *channels* to schedule creation of a burst for which the *core burst* will start at a specified delay taken after the *start time* of the next strobe occurrence of the specified strobe source, as depicted in the following figure:

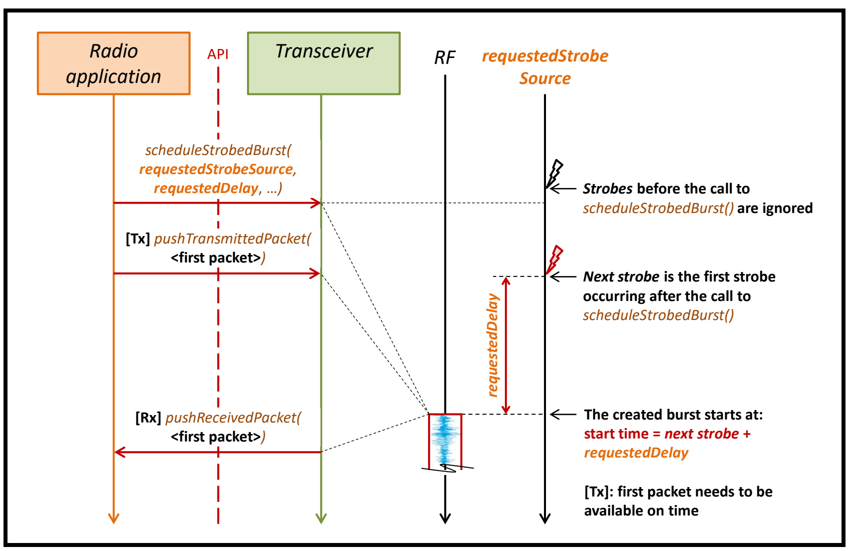

Figure 44 Principle of *scheduleStrobedBurst()*

The standard strobe sources **are specified by** the following table:

Name	Description
ApplicationStrobe	Strobes delivered by the *radio application* using the **ApplicationStrobe** interface.
TimeRef_PPS	Strobes delivered by the PPS signal of a wired time reference.
GNSS_PPS	Strobes delivered by the PPS signal of a GNSS system.
UserStrobe1	User-defined strobe 1.
UserStrobe2	User-defined strobe 2.
UserStrobe3	User-defined strobe 3.
UserStrobe4	User-defined strobe 4.

Table 6 Specification of strobe sources

Additional *strobe sources* can be implemented if required by usage context.

3.1.6.1.2 Associated properties

CREATION_STORAGE (see section 4.8) specifies the maximum number of calls to *creation operations* that *channels* can store in advance, including calls to *scheduleStrobedBurst()*.

STROBED_MILT (see section 4.13) specifies the minimum invocation lead time for correct real-time usage of the operation.

STROBED_WCET (see section 4.15) specifies the worst-case execution time of the primitive.

3.1.6.1.3 Declaration

The declaration of the operation **is specified as**:

```
void scheduleStrobedBurst(
    in StrobeSource requestedStrobeSource,
    in Delay requestedDelay,
    in BlockLength requestedLength);
```

3.1.6.1.4 Parameters

Name	Type	Description
requestedStrobeSource	***StrobeSource*** (see § 3.4.14)	Specifies the *strobe source* to be used.
requestedDelay	***Delay*** (see § 3.4.7)	Specifies the delay between the *next strobe occurrence* on *strobe source* and *start time* of the burst to create.
requestedLength	***BlockLength*** (see § 3.4.3)	Number of *baseband samples* to be processed during the *processing phase* associated to the *burst*: ▪ If equal to **UndefinedBlockLength**: specifies an *undefined* value, ▪ If not equal to **UndefinedBlockLength**: specifies the *number* of *baseband samples* of the *baseband block* to be processed during **PROCESSING** (see section 2.3.1).

Table 7 Specification of *scheduleStrobedBurst()* parameters

The *parameters validity properties* **are specified as** (see section 4.7):

- For *requestedStrobeSource*: **STROBE_SOURCES**,
- For *requestedDelay*: **MIN_FROM_STROBE** and **MAX_FROM_STROBE**,
- For *requestedLength*: **MIN_BLOCK_LENGTH** and **MAX_BLOCK_LENGTH**.

3.1.6.1.5 Return value

None.

3.1.6.1.6 Originator

Radio application.

3.1.6.1.7 Exceptions

The *exceptions* of the *operation* **are specified as** (see section 3.2):

- **StrobeSource**,
- **MinFromStrobe** and **MaxFromStrobe**,
- **MinBlockLength** and **MaxBlockLength**.

3.1.6.1.8 Behavior requirements

An *active instance* of **StrobedCreation** **shall**, on a call to *scheduleStrobedBurst()*, handle the *exceptions* of the *operation* as specified in section 3.2.

An *active instance* of **StrobedCreation** **shall**, on a call to *scheduleStrobedBurst()* that raised no exception:

- If **CREATION_STORAGE** calls (see section 4.8) are stored, wait until storage becomes available,
- Store the call for later usage by CreationControl (see section 2.3.2),
- Return the call to the *radio application*.

3.1.7 Transceiver::BurstControl::Termination

3.1.7.1 setBlockLength operation

3.1.7.1.1 Overview

setBlockLength() specifies the length of *baseband block* applicable for termination of an ongoing *processing phase*.

3.1.7.1.2 Associated properties

BLOCK_LENGTH_MILT (see section 4.13) specifies the minimum invocation lead time for correct real-time usage of the operation.

BLOCK_LENGTH_WCET (see section 4.15) specifies the worst-case execution time of the primitive.

3.1.7.1.3 Declaration

The declaration of the operation **is specified as**:

```
void setBlockLength(
    in BlockLength requestedLength);
```

3.1.7.1.4 Parameters

Name	Type	Description
requestedLength	***BlockLength*** (see § 3.4.3)	Number of *baseband samples* to be processed during **PROCESSING** (see section 2.3.1).

Table 8 Specification of *setBlockLength()* parameters

The *parameters validity properties* **are specified as** (see section 4.7):

- For *requestedLength*: **`MIN_BLOCK_LENGTH`** and **`MAX_BLOCK_LENGTH`**.

3.1.7.1.5 Return value

None.

3.1.7.1.6 Originator

Radio application.

3.1.7.1.7 Exceptions

The *exceptions* of the *operation* **are specified as** (see section 3.2):

- **`NoOngoingProcessing`**,
- **`MinBlockLength`** and **`MaxBlockLength`**.

3.1.7.1.8 Behavior requirements

An *active instance* of **`Termination`** **shall**, on a call to *setBlockLength()*, handle the *exceptions* of the *operation* as specified in section 3.2.

An *active instance* of **`Termination`** **shall**, on a call to *setBlockLength()* that raised no exception:

- Set value of **`applicableBurstLength`** to value of *requestedLength*,
- Notify the **`PROCESSING`** state of **`Channels`** of availability of a new **`applicableBurstLength`** value,
- Return the call to the *radio application*.

3.1.7.2 stopBurst operation

3.1.7.2.1 Overview

stropBurst() commands an ongoing *processing phase* to immediately terminate.

3.1.7.2.2 Associated properties

None.

3.1.7.2.3 Declaration

The declaration of the operation **is specified as**:

```
void stopBurst();
```

3.1.7.2.4 Parameters

None.

3.1.7.2.5 Return value

None.

3.1.7.2.6 Originator

Radio application.

3.1.7.2.7 Exceptions

The *exceptions* of the *operation* **are specified as** (see section 3.2):

- `NoOngoingProcessing`.

3.1.7.2.8 Behavior requirements

An *active instance* of `Termination` **shall**, on a call to *setBlockLength()*, handle the *exceptions* of the *operation* as specified in section 3.2.

An *active instance* of `Termination` **shall**, on a call to *stopBurst()* that raised no exception:

- Set value of `applicableBurstLength` to the value enabling fastest possible termination of the ongoing *processing phase*,
- Notify the `PROCESSING` state of `Channels` of availability of a new `applicableBurstLength` value,
- Return the call to the *radio application*.

3.1.8 Transceiver::BasebandSignal::SamplesReception

3.1.8.1 pushRxPacket Operation

3.1.8.1.1 Overview

pushRxPacket() provides the *radio application* with the next packet of an *Rx block* received by one *Rx channel*, as depicted in following figure:

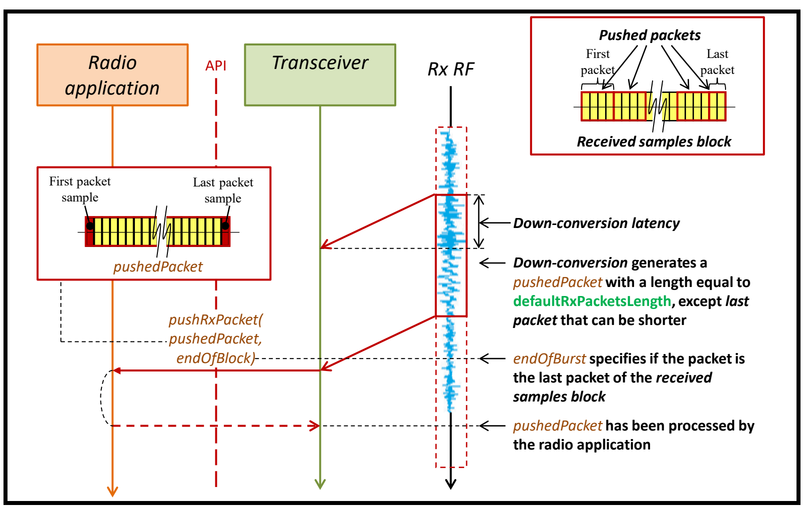

Figure 45 Principle of *pushRxPacket()*

Rx packets are sent by the *Rx channel* until the *last processed sample* (see section 2.3.1.1.5) has been transferred.

The first packet is sent after the `ProcessingStart` transition (see section 2.3.1).

A boolean flag specifies to the *radio application* that a received packet is the last packet of the received samples block. The next received packet is the first packet of the next received samples block.

3.1.8.1.2 Associated properties

RX_MIN_BASEBAND_LEVEL and **RX_MAX_BASEBAND_LEVEL** (see section 4.10) specify the interval into which the level of *baseband signal* fits.

RX_META_DATA (see section 4.5) specifies if meta-data is associated to *Rx packets*.

RX_PACKET_WCET (see section 4.15) specifies the worst-case execution time of the primitive for correct real-time operation of the *transceiver instance*.

3.1.8.1.3 Declaration

The declaration of the operation **is specified as**, if **RX_META_DATA** is equal to **FALSE**:

```
void pushRxPacket(
    in BasebandPacket rxPacket,
    in boolean endOfBlock);
```

The declaration of the operation **is specified as**, if **RX_META_DATA** is equal to **TRUE**:

```
void pushRxPacket(
    in BasebandPacket rxPacket,
    in boolean endOfBlock,
    in RxMetaData rxMetaData);
```

3.1.8.1.4 Parameters

Name	Type	Description
rxPacket	***BasebandPacket*** (see § 0)	The transferred *Rx packet* within the *Rx block*.
endOfBlock	***boolean***	Specifies if *rxPacket* is the last packet of the *Rx block*.
rxMetaData If **RX_META-DATA** is equal to **TRUE**.	***RxMetaData*** (see § 3.4.12)	Specifies the user-defined *meta-data* associated to the *Rx packet*.

Table 9 Specification of *pushRxPacket*() parameters

No *parameters validity property* is specified for *use services*.

3.1.8.1.5 Return value

None.

3.1.8.1.6 Originator

Rx channel.

3.1.8.1.7 Exceptions

Not applicable to a *use service*.

3.1.8.1.8 Behavior requirements

nbrFullPackets and *tailPacketLength* **are defined as**, respectively, the quotient and the remainder of the Euclidean division of `applicableBlockLength` by `applicableRxPacketsLength`.

Active instances of `SamplesReception` **shall** transfer the *Rx block* with a succession of *nbrFullPackets* calls to *pushRxPacket()*, with the length of *rxPacket* equal to `applicableRxPacketsLength`.

Active instances of `SamplesReception` **shall**, if *tailPacketLength* is greater than `0`, make a last call to *pushRxPacket()* with the length of *rxPacket* equal to *tailPacketLength*.

Active instances of `SamplesReception` **shall** set the value of *endOfBlock* as follows:

- `false`: for all calls to *pushRxPacket()* except the last one,
- `true`: for the last call to *pushRxPacket()*.

Active instances of `SamplesReception` **shall** wait for the *radio application* to return the previous call to *pushRxPacket()* before making a next call to *pushRxPacket()*.

3.1.9 Transceiver::BasebandSignal::SamplesTransmission

3.1.9.1 pushTxPacket Operation

3.1.9.1.1 Overview

pushTxPacket() provides a *Tx channel* with the next packet of a *Tx forwarded block* to be stored prior to up-conversion, as depicted in following figure:

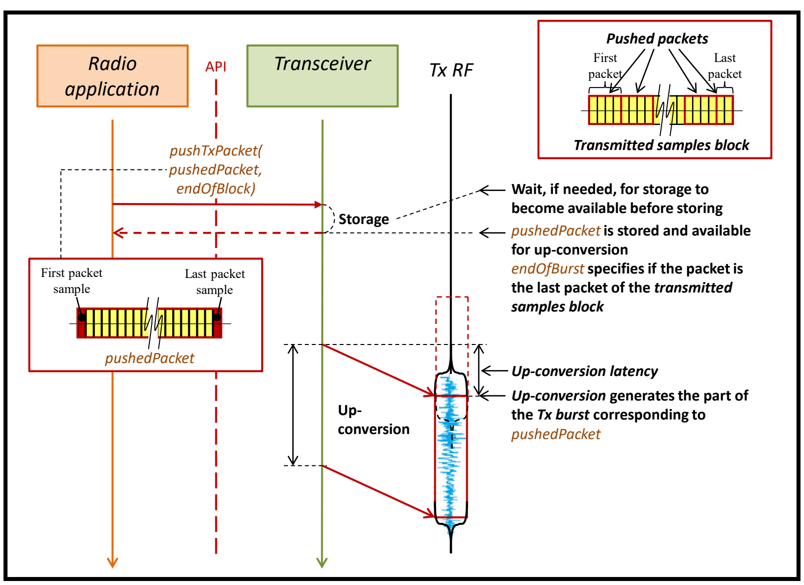

Figure 46 Principle of *pushTxPacket()*

The length of each packet is determined by the *radio application*, and can vary from one packet to another down to a single sample packet.

The first packet of the first *Tx forwarded block* is the first packet after **ResetCompleted** transition (see section 2.3.1.2.1).

A flag specifies to the *Tx channel* the last packet of the *Tx forwarded block*. Next transmitted packet is the first packet of the next *Tx forwarded block*.

3.1.9.1.2 Associated properties

TX_META_DATA (see section 4.5) specifies if meta-data is associated to *Tx packets*.

TX_BASEBAND_STORAGE (see section 4.8) specifies the number of *baseband samples* that a *transceiver* can store in advance of their usage by up-conversion.

TX_PACKET_MILT (see section 4.13) specifies the minimum invocation lead time for correct real-time usage of the operation.

TX_MIN_BASEBAND_LEVEL and **TX_MAX_BASEBAND_LEVEL** (see section 4.10) specify the interval into which the level of *baseband signal* must belong for correct *Rx channel* operation.

TX_PACKET_WCET (see section 4.15) specifies the worst-case execution time of the primitive.

3.1.9.1.3 Declaration

The declaration of the operation **is specified as**, if **TX_META_DATA** is equal to **FALSE**:

```
void pushTxPacket(
    in BasebandPacket txPacket,
    in boolean endOfBlock);
```

The declaration of the operation **is specified as**, if **TX_META_DATA** is equal to **TRUE**:

```
void pushTxPacket(
    in BasebandPacket txPacket,
    in boolean endOfBlock,
    in TxMetaData txMetaData);
```

3.1.9.1.4 Parameters

Name	Type	Description
txPacket	***BasebandPacket*** (see § 0)	The transferred *Tx packet*.
endOfBlock	***boolean***	Specifies that *txPacket* is the last packet of the *Tx forwarded block*.
txMetaData If **TX_META-DATA** is equal to **TRUE**.	***TxMetaData*** (see § 3.4.12)	Specifies the user-defined *meta-data* associated to the *Tx packet*.

Table 10 Specification of *pushTxPacket()* parameters

The *parameters validity properties* **are specified as** (see section 4.7):

- For length of *txPacket*: **MAX_PACKETS_LENGTH**.

3.1.9.1.5 Return value

None.

3.1.9.1.6 Originator

Radio application.

3.1.9.1.7 Exceptions

The *exceptions* of the *operation* **are specified as** (see section 3.2):

- **MaxTxPacketsLength**,
- **TxPacketsMILT**.

3.1.9.1.8 Behavior requirements

Active instances of **SamplesTransmission** **shall**, on a call to *pushTxPacket()*, handle the *exceptions* of the *operation* as specified in section 3.2.

Active instances of **SamplesTransmission** **shall**, on a call to *pushTxPacket()* that raised no exception:

- Wait, if needed, until all *baseband samples* of a *previous burst* have entered up-conversion,
- Wait, if storage is saturated, for consumption by up-conversion of previously stored samples to free storage capacity,
- Store the samples of *txPacket* for later usage by *up-conversion*,
- Depending on value of *endOfBlock*:
  - **false**: *Tx forwarded block* is not ended,
  - **true**: *Tx forwarded block* is ended, the *last sample* of *txPacket* is the last sample of the *Tx forwarded block*,
- Return the call to the *radio application*.

A *channel* **shall** be capable to store up to **TX_BASEBAND_STORAGE** (see section 4.6) *baseband samples*.

3.1.10 Transceiver::BasebandSignal::RxPacketsLengthControl

3.1.10.1 setRxPacketsLength operation

3.1.10.1.1 Overview

setRxPacketsLength() provides *Rx channels* with the size of *received packets* to be used at creation of forthcoming *Rx bursts*.

3.1.10.1.2 Associated properties

RX_PACKETS_LENGTH_WCET (see section 4.15) specifies the worst-case execution time of the primitive.

3.1.10.1.3 Declaration

The declaration of the operation **is specified as**:

```
void setRxPacketsLength(
    in PacketLength requestedLength);
```

3.1.10.1.4 Parameters

Name	Type	Description
requestedLength	**`PacketLength`** (see § 3.4.12)	Specifies the new value for **`applicableRxPacketsLength`** attribute (see § 3.3.1.2).

Table 11 Specification of *setRxPacketsLength()* parameters

The *parameters validity properties* **are specified as** (see section 4.7):

- For *requestedLength*: **`MAX_PACKETS_LENGTH`**.

3.1.10.1.5 Return value

None.

3.1.10.1.6 Originator

Radio application.

3.1.10.1.7 Exceptions

The *exceptions* of the *operation* **are specified as** (see section 3.2):

- **`MaxRxPacketsLength`**.

3.1.10.1.8 Behavior requirements

An *active instance* of **`RxPacketsLengthControl`** **shall**, on a call to *setRxPacketsLength()*, handle the *exceptions* of the *operation* as specified in section 3.2.

An *active instance* of **`RxPacketsLengthControl`** **shall**, on a call to *setRxPacketsLength()* that raised no exception:

- Sets value of **`applicableRxPacketsLength`** attribute (see section 3.3.1.2) to value of *requestedLength* parameter,
- Return the call to the *radio application*.

3.1.11 Transceiver::Tuning::InitialTuning

3.1.11.1 setTuning Operation

3.1.11.1.1 Overview

setTuning() commands the *channels* to store a *tuning parameters set* (composed of *tuning preset, carrier frequency* (f_c) and *gain* (G), see section 1.1.4) than will be later applied to the *tuned burst*, as depicted in the following figure:

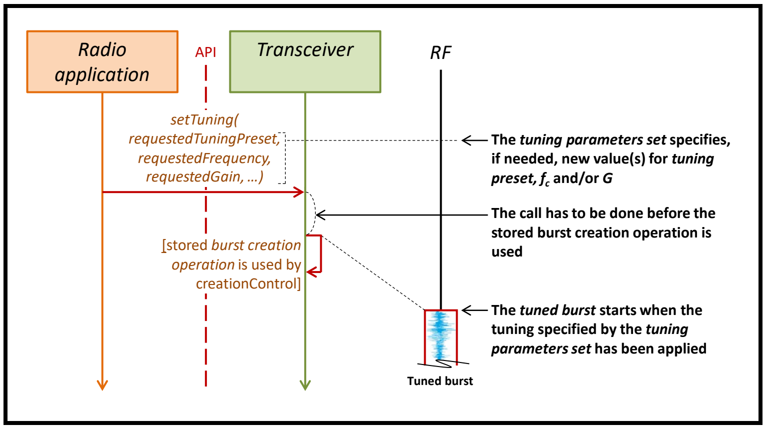

Figure 47 Principle of *setTuning()*

The call to *setTuning()* for a given *burst* needs to done, if needed, before **CreationControl** enters in **INITIATING** state for the considered burst (see section 2.3.2).

3.1.11.1.2 Associated properties

TUNING_ASSOCIATION (see section 4.3) specifies how **CreationControl** (see section 2.3.2) associates stored tuning calls to created bursts during **INITIATING** state.

TUNING_STORAGE (see section 4.8) specifies the maximum number of calls to *setTuning() channels* can store in advance.

TUNING_MILT (see section 4.13) specifies the minimum invocation lead time in advance of the call to the *creation operation* of the tuned burst.

TUNING_WCET (see section 4.15) specifies the worst-case execution time of the primitive.

3.1.11.1.3 Declaration

The declaration of the operation **is specified as**:

```
void setTuning(
    in TuningPreset requestedPreset,
    in CarrierFreq requestedFrequency,
    in Gain requestedGain,
    in BurstNumber requestedBurstNumber);
```


3.1.11.1.4 Parameters

Name	Type	Description
requestedPreset	**TuningPreset** (see § 3.4.14)	*Tuning preset* to be applied under control of *burst creation* during a **TUNING** state: ▪ If equal to **UndefinedTuningPreset**: specifies to reuse the previously active *tuning preset*, ▪ If not equal to **UndefinedTuningPreset**: specifies the *tuning preset* to apply.
requestedFrequency	**CarrierFreq** (see § 3.4.6)	*Carrier frequency* (f_c) to be applied under control of *burst creation* during a **TUNING** state: ▪ If equal to **UndefinedCarrierFreq**: specifies to reuse the previously active *carrier frequency*, ▪ If not equal to **UndefinedCarrierFreq**: specifies the *carrier frequency* to apply.
requestedGain	**Gain** (see § 3.4.10)	*Gain* (*G*) to be applied under control of *burst creation* during a **TUNING** state: ▪ If equal to **UndefinedGain**: specifies to reuse the previously active *gain*, ▪ If not equal to **UndefinedGain**: specifies the *gain* to apply.
requestedBurstNumber	**BurstNumber** (see § 3.4.5)	Specifies a burst number for *burst creation* to determine the tuned burst for the specified *tuning parameters set*, depending on value of TUNING_ASSOCIATION: ▪ If equal to **sequential**: the value is ignored, ▪ If equal to **burstReferencing**: the specified number of the burst for which the specified tuning parameters set applies.

Table 12 Specification of *setTuning()* parameters

The *parameters validity properties* **are specified as** (see section 4.7):

- For *requestedPreset*: **MAX_TUNING_PRESET**,
- For *requestedFrequency*: **MIN_CARRIER_FREQ** and **MAX_CARRIER_FREQ**,
- For *requestedGain*: **MIN_GAIN** and **MAX_GAIN**.

3.1.11.1.5 Return value

None.

3.1.11.1.6 Originator

Radio application.

3.1.11.1.7 Exceptions

The *exceptions* of the *operation* **are specified as** (see section 3.2):

- **MaxTuningPreset**,
- **MinCarrierFreq** and **MaxCarrierFreq**,
- **MinGain** and **MaxGain**,
- **TuningMILT**.

3.1.11.1.8 Behavior requirements

An *active instance* of **InitialTuning** **shall**, on a call to *setTuning()*, handle the *exceptions* of the *operation* as specified in section 3.2.

An *active instance* of **InitialTuning** **shall**, on a call to *setTuning()* that raised no exception:

- Wait, if storage is saturated, for usage of a previously stored call to free capacity,
- Store the call by order of arrival for later usage by **CreationControl** (see section 2.3.2),
- Return the call to the *radio application*.

A *channel* **shall** be capable to store up to **TUNING_STORAGE** (see section 4.8) *setTuning()* calls.

3.1.12 Transceiver::Tuning::Retuning

3.1.12.1 retune Operation

3.1.12.1.1 Overview

retune() commands the *channels* to change the tuning during an ongoing *processing phase*, specifying the delay to take from the *start time* of the burst before starting to retune, as depicted in following figure:

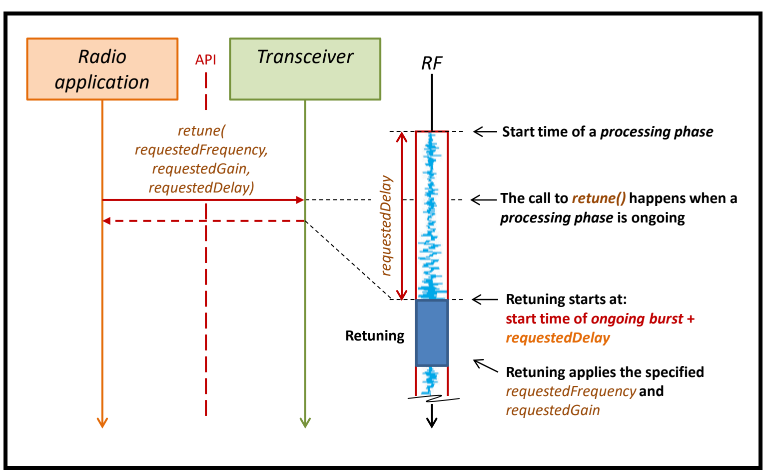

Figure 48 Principle of *retune()*

An undefined *delay* specifies retuning to take place immediately after the call to *retune()*.

3.1.12.1.2 Associated properties

RETUNING_DURATION (see section 4.8) specifies the maximum duration of RETUNING state (see section 2.3.4).

RETUNING_MILT (see section 4.13) specifies the minimum invocation lead time for correct real-time usage of the operation.

RETUNING_WCET (see section 4.15) specifies the worst-case execution time of the primitive.

3.1.12.1.3 Declaration

The declaration of the operation **is specified as**:

```
void retune(
    in CarrierFreq requestedFrequency,
    in Gain requestedGain,
    in Delay requestedDelay);
```

3.1.12.1.4 Parameters

Name	Type	Description
requestedFrequency	**CarrierFreq** (see § 3.4.6)	*Carrier frequency* (f_c) to be applied by *channels* during the scheduled RETUNING state: ▪ If equal to UndefinedCarrierFreq: specifies to reuse the previously active *carrier frequency*, ▪ If not equal to UndefinedCarrierFreq: specifies the *carrier frequency* to apply.
requestedGain	**Gain** (see § 3.4.10)	*Gain* (*G*) to be applied by *channels* during the scheduled RETUNING state: ▪ If equal to UndefinedGain: specifies to reuse the previously active *gain*, ▪ If not equal to UndefinedGain: specifies the *gain* to apply.
requestedDelay	**Delay** (see § 3.4.7)	Delay to take after the *start time* of the ongoing *processing phase* for triggering the RetuningStart transition: ▪ If equal to UndefinedDelay: specifies that the RetuningStart transition is triggered immediately, ▪ If not equal to UndefinedDelay: specifies the applicable delay.

Table 13 Specification of *retune()* parameters

The *parameters validity properties* **are specified as** (see section 4.7):

- For *requestedFrequency*: MIN_CARRIER_FREQ and MAX_CARRIER_FREQ,
- For *requestedGain*: MIN_GAIN and MAX_GAIN,
- For *requestedDelay*: MIN_FROM_ONGOING and MAX_FROM_ONGOING.

3.1.12.1.5 Return value

None.

3.1.12.1.6 Originator

Radio application.

3.1.12.1.7 Exceptions

The *exceptions* of the *operation* **are specified as** (see section 3.2):

- `NoOngoingProcessing`,
- `MinCarrierFreq` and `MaxCarrierFreq`,
- `MinGain` and `MaxGain`,
- `MinFromOngoing` and `MaxFromOngoing`,
- `RetuningMILT`.

3.1.12.1.8 Behavior requirements

An *active instance* of `Retuning` **shall**, on a call to *retune()*, handle the *exceptions* of the *operation* as specified in section 3.2.

An *active instance* of `Retuning` **shall**, on a call to *retune()* that raised no exception, with value of *requestedDelay* equal to `UndefinedDelay`:

- Return the call to *retune()* to the *radio application*,
- Trigger the `RetuningStart` transition (see section 2.3.4) immediately after.

An *active instance* of `Retuning` **shall**, on a call to *retune()* that raised no exception, with value of *requestedDelay* not equal to `UndefinedDelay`:

- Return the call to *retune()* to the *radio application*,
- Trigger the `RetuningStart` transition (see section 2.3.4) at *start time* of the ongoing *processing phase* plus value of *requestedDelay*.

A *channel* **shall**, during `RETUNING` state, act on the *carrier frequency* according to *requestedFrequency* value:

- If equal to `UndefinedCarrierFreq`: keep the *carrier frequency* unchanged.
- If not equal to `UndefinedCarrierFreq`: apply *requestedFrequency* as the new *carrier frequency*,

A *channel* **shall**, during `RETUNING` state, act on the *gain* according to *requestedGain* value:

- If equal to `UndefinedGain`: keep the *gain* unchanged,
- If not equal to `UndefinedGain`: apply *requestedGain* as the new *gain*,

3.1.13 Transceiver::Notifications::Events

3.1.13.1 notifyEvent Operation

3.1.13.1.1 Overview

An *event* **is defined as** occurrence of a condition related to operation of a *channel*.

notifyEvent() informs the *radio application* that a defined *event* has occurred as depicted in following figure:

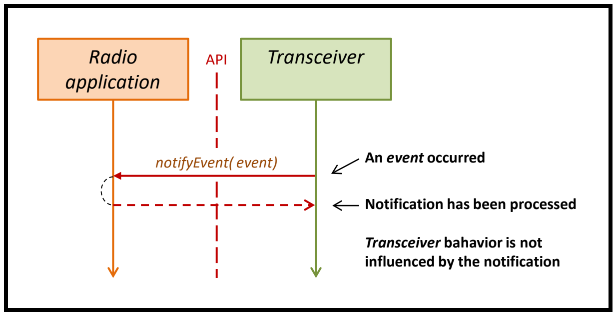

Figure 49 Principle of *notifyEvent()*

Events **are specified by** the following table:

Name / <eventName>	Description	See §
eventProcessingStart	Applies to: all *channels*. Condition: *channels* make a **ProcessingStart** transition.	2.3.1
eventProcessingStop	Applies to: all *channels*. Condition: *channels* make a **ProcessingStop** transition.	2.3.1
eventSilenceStart	Applies to: *channels* capable of *radio silence*. Condition: *channels* make a **SilenceStart** transition.	2.3.2
eventSilenceStop	Applies to: *channels* capable of *radio silence*. Condition: *channels* make a **SilenceStop** transition.	2.3.2

Table 14 Specification of events

3.1.13.1.2 Associated properties

EVENTS (see section 4.40) specifies, for each *event*, if *event notification* has to be performed.

EVENTS_WCET (see section 4.15) specifies the worst-case execution time of the primitive for correct real-time operation of the *transceiver instance*.

3.1.13.1.3 Declaration

The declaration of the operation **is specified as**:

```
void notifyEvent(
    in Event notifiedEvent);
```

3.1.13.1.4 Parameters

Name	Type	Description
notifiedEvent	***Event*** (see §3.4.9)	Enumerated value specifying the notified *event*.

Table 15 Specification of *notifyEvent()* parameters

No *parameters validity property* is specified for *use services*.

3.1.13.1.5 Return value

None.

3.1.13.1.6 Originator

Transceiver.

3.1.13.1.7 Exception

Not applicable to *use services*.

3.1.13.1.8 Behavior requirements

Channels with an *active instance* of **Events** **shall**, when **<eventName>** happens and **EVENTS.<eventName>** is equal to **true**, call *notifyEvent()* with *notifiedEvent* equal to **<eventName>**.

Channels with an *active instance* of **Events** **shall**, if *channels* have been set in *radio silence* by another agent than the *radio application* when **INITIALIZATION** terminates, call *notifyEvent()* with *notifiedEvent* equal to **eventSilenceStart**.

3.1.14 Transceiver::Notifications::Errors

3.1.14.1 notifyError Operation

3.1.14.1.1 Overview

An *error* **is defined as** an abnormal situation related to *channels* internal execution errors.

notifyError() informs the *radio application* that a defined *error* (see section 3.2) has occurred as depicted in following figure:

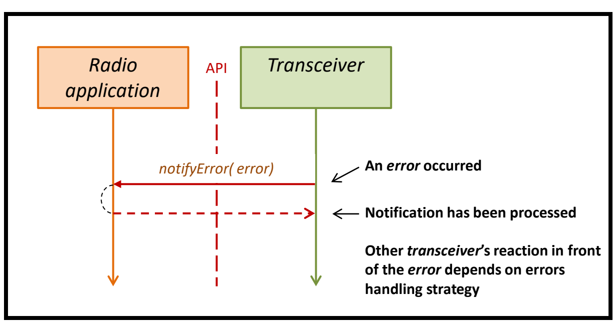

Figure 50 Principle of *notifyError()*

Errors **are specified by** the following table:

Name / <errorName>	Specification	See §
error1stSampleDelayed	Applies to: *Tx channels* with at least one *active instance* of a *timely creation service*. Condition: the *first sample* of a *Tx forwarded block* is not available before *activation time*.	2.3.2
error1stSampleTimeout	Applies to: *Tx channels* with at least one *active instance* of a *timely creation service*, if **ERRORS.err1stSampleDelayed.reaction** is equal to **mitigating**. Condition: during a *burst creation*, the *first sample* of a *Tx forwarded block* is not available once **1ST_SAMPLE_TIMEOUT** nanoseconds have elapsed after *activation time*.	2.3.2
errorBurstOverlap	Applies to: *channels* with at least one *active instance* of a *timely creation service*. Condition: *activation time* of a *burst under creation* and termination time of the *previous burst* do not enable respect value of **INTER-PROCESS**.	2.3.2 4.8
errorRxOverflow	Applies to: *Rx channels*. Condition: the *radio application* did not return a *pushRxPacket()* call when *Rx channels* have to make the next call.	3.1.7
errorShorterTxBlock	Applies to: *Tx channels*. Condition: a *Tx forwarded block* is ended (value of *endOfBlock* in a call to *pushTxPacket()* is set to **true**) and *requestedLength* is equal to **UndefinedBlockLength** or length of the *baseband block* is smaller than a defined value of *requestedLength*.	2.3.1
errorTxUnderflow	Applies to: *Tx channels*. Condition: *baseband samples* are not available early enough for a *Tx channel* to proceed with *up-conversion* during a **PROCESSING** state.	3.1.8
errorTuningDelayed	Applies to: *channels* with an *active instance* of **InitialTuning**. Condition: during a *burst creation*, the **TuningStop** transition has not occurred at the time required for **ProcessingStart** transition to satisfy the *start time*.	3.1.11
errorTuningTimeout	Applies to: *channels* with an *active instance* of **InitialTuning**, if **ERRORS.errTuningDelayed.reaction** is equal to **mitigating**. Condition: during a *burst creation*, **TuningStop** transition has not occurred once **TUNING_TIMEOUT** nanoseconds elapsed after the *activation time*.	2.3.2

Table 16 Specification of errors

3.1.14.1.2 Associated properties

ERRORS (see section 4.4) specifies, for each *error*, how it is handled by *active instances* of **Errors**:

- Applicable behavior when the *error* happens,
- If *error notification* has to be performed.

ERRORS_WCET (see section 4.15) specifies the worst-case execution time of the primitive for correct real-time operation of the *transceiver instance*.

3.1.14.1.3 Declaration

The declaration of the operation **is specified as**:

```
void notifyError(
    in Error notifiedError);
```

3.1.14.1.4 Parameters

Name	Type	Description
notifiedError	**Error** (see § 3.4.8)	Specifies the notified *error*.

Table 17 Specification of *notifyError()* parameters

No *parameters validity property* is specified for *use services*.

3.1.14.1.5 Return value

None.

3.1.14.1.6 Originator

Transceiver.

3.1.14.1.7 Exceptions

Not applicable to *use services*.

3.1.14.1.8 Behavior requirements

Error notification of **<errorName>** **is defined as** a call to *notifyError()*, independently of other *channels* operation, with *notifiedError* parameter equal to **<errorName>**.

Channels with an *active instance* of **Errors**, when **<errorName>** happens and **ERRORS.<errorName>.reaction** is equal to **fatal**, have *unspecified* behavior.

Channels with an *active instance* of **Errors** **shall**, when **<errorName>** happens and **ERRORS.<errorName>.reaction** is equal to **resetting**:

- Trigger a **RuntimeReset** transition,
- If **ERRORS.<errorName>.isNotified** is equal to **true**, perform *error notification*.

Channels with an *active instance* of `Errors` **shall**, when `<errorName>` happens and `ERRORS.<errorName>.reaction` is equal to `mitigation`:

- Perform the *error mitigation behavior* specified in Table 18,
- If `ERRORS.<errorName>.isNotified` is equal to `true`, perform *error notification*.

Errors mitigation behaviors **are specified by** the following table:

Error name	Error mitigation behavior
`error1stSampleDelayed`	Wait until the *first baseband sample* is available, then make a `ProcessingStart` transition (entry in `PROCESSING` state, see section 2.3.1).
`error1stSampleTimeout`	*Unspecified.*
`errorBurstOverlap`	Call *setBlockLength()* with *requestedLength* shortening the length of previous burst so that its *termination time* is smaller than the *tuning time* of the *burst under creation*.
`errorRxOverflow`	Drop the *baseband samples* delivered by *down-conversion* until the pending call to *pushRxPacket()* returns.
`errorShorterTxBlock`	Call *setBlockLength()* with *requestedLength* equal to the length of the *Tx forwarded block*.
`errorTxUnderflow`	Pad missing *baseband samples* with *unspecified* samples until new *baseband samples* are available.
`errorTuningDelayed`	Wait until `TuningStop` transition, then make a `ProcessingStart` transition (entry in `PROCESSING` state, see section 2.3.1).
`errorTuningTimeout`	*Unspecified.*

Table 18 Specification of errors mitigation behaviors

3.1.15 Transceiver::GainControl::GainChanges

3.1.15.1 indicateGain Operation

3.1.15.1.1 Overview

indicateGain() provides the *radio application* with a new value of *gain* decided by *channels* during a *processing phase* as depicted in following figure:

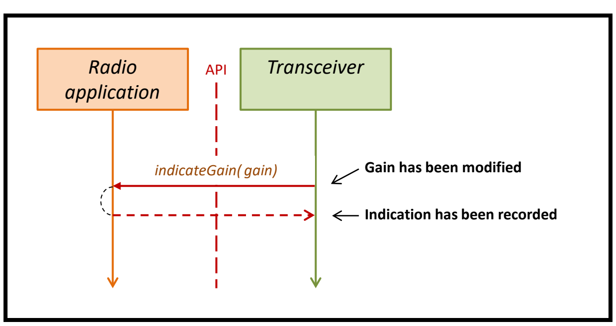

Figure 51 Principle of *indicateGain()*

3.1.15.1.2 Associated properties

GAIN_CHANGE_WCET (see section 4.15) specifies the worst-case execution time of the primitive for correct real-time operation of the *transceiver instance*.

3.1.15.1.3 Declaration

The declaration of the operation **is specified as**:

```
void indicateGain(
    in Gain newGain,
    in SampleNumber firstValidSample
);
```

3.1.15.1.4 Parameters

Name	Type	Description
newGain	**Gain** (see § 3.4.10)	Specifies the new value of *gain*.
firstValidSample	**SampleNumber** (see § 3.4.14)	Sample number of the first sample in the *Rx block* after which the tuning is stable again.

Table 19 Specification of *indicateGain()* parameters

No *parameters validity property* is specified for *use services*.

3.1.15.1.5 Return value

None.

3.1.15.1.6 Originator

Transceiver.

3.1.15.1.7 Exceptions

Not applicable to *use services*.

3.1.15.1.8 Behavior requirements

An *active instance* of **GainChanges** **shall** indicate each new value of *gain* using *indicateGain()*.

3.1.16 Transceiver::GainControl::GainLocking

3.1.16.1 lockGain Operation

3.1.16.1.1 Overview

lockGain() commands *Rx channels* to lock the applied *Rx gain*, that becomes not modifiable by *AGC*.

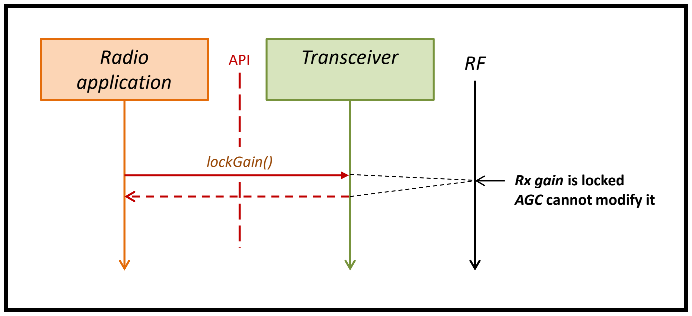

Figure 52 Principle of *lockGain()*

3.1.16.1.2 Associated properties

`LOCK_GAIN_WCET` (see section 4.15) specifies the worst-case execution time of the primitive.

3.1.16.1.3 Declaration

The declaration of the operation **is specified as**:

```
void lockGain();
```

3.1.16.1.4 Parameters

None.

3.1.16.1.5 Returned value

None.

3.1.16.1.6 Originator

Radio application.

3.1.16.1.7 Exceptions

The *exceptions* of the *operation* **are specified as** (see section 3.2):

- `NoOngoingProcessingException`.

3.1.16.1.8 Behavior requirements

An *active instance* of `GainLocking` **shall**, on a call to *lockGain()*:

- Lock value of *Rx gain* at its current value independently of *AGC* operation,
- Return the call to the *radio application*.

3.1.16.2 unlockGain Operation

3.1.16.2.1 Overview

unlockGain() commands *Rx channels* to unlock *Rx gain*, that becomes subject to modifications under control of *AGC*.

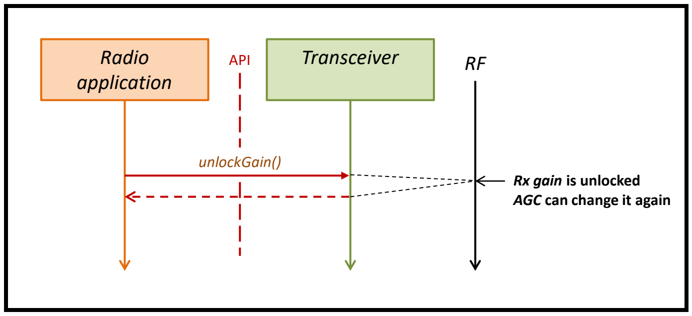

Figure 53 Principle of *unlockGain()*

3.1.16.2.2 Associated properties

UNLOCK_GAIN_WCET (see section 4.15) specifies the *maximum processing time* for correct joint real-time operation of *radio application* and *transceiver*.

3.1.16.2.3 Declaration

The declaration of the operation **is specified as**:

```
void unlockGain();
```

3.1.16.2.4 Parameters

None.

3.1.16.2.5 Returned value

None.

3.1.16.2.6 Originator

Radio application.

3.1.16.2.7 Exceptions

The *exceptions* of the *operation* **are specified as** (see section 3.2):

- **NoOngoingProcessingException**.

3.1.16.2.8 Behavior requirements

An *active instance* of **GainLocking** **shall**, on a call to *unlockGain()*:

- Enable *Rx gain* to be modified by *AGC*,
- Return the call to the *radio application*.

3.1.17 Transceiver::TransceiverTime::TimeAccess

3.1.17.1 getCurrentTime Operation

3.1.17.1.1 Overview

getCurrentTime() commands the *channels* to return the value of *transceiver time* corresponding to return time of the call, as depicted in following figure:

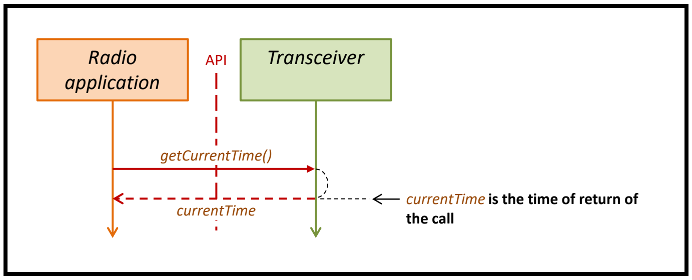

Figure 54 Principle of *getCurrentTime()*

3.1.17.1.2 Associated properties

CURRENT_TIME_ACC (see section 4.12) specifies the accuracy of the returned *currentTime* value.

CURRENT_TIME_WCET (see section 4.15) specifies the worst-case execution time of the primitive.

3.1.17.1.3 Declaration

The declaration of the operation **is specified as**:

```
void getCurrentTime(
    out TimeSpec currentTime);
```

3.1.17.1.4 Parameters

Name	Type	Description
currentTime	**TimeSpec** (see § 3.4.16)	Value of *transceiver time* when *getCurrentTime()* returns.

Table 20 Specification of *getCurrentTime()* parameters

No *parameters validity property* is associated to *out* parameters.

3.1.17.1.5 Return value

None.

3.1.17.1.6 Originator

Radio application.

3.1.17.1.7 Exceptions

None.

3.1.17.1.8 Behavior requirements

An *active instance* of **TimeAccess** **shall**, on a call to *getCurrentTime()*:

- Set the return value of *currentTime* to a value belonging to value of *transceiver time* when *getCurrentTime()* returns ± **CURRENT_TIME_ACC**,
- Return the call to the *radio application*.

3.1.17.2 getLastStartTime Operation

3.1.17.2.1 Overview

getLastStartTime() commands the *channels* to return the value of *transceiver time* corresponding to the start time of the last burst created by the *channels* for which *getLastStartTime()* is called, and to return its *burst number*, as depicted in following figure:

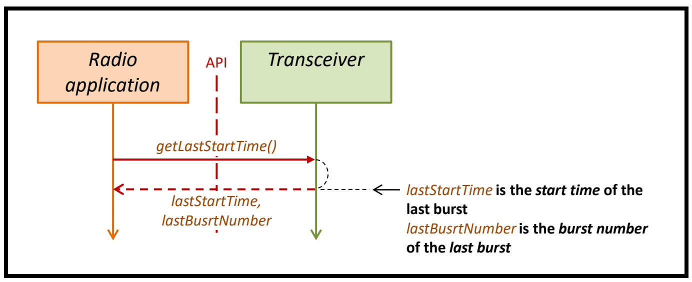

Figure 55 Principle of *getLastStartTime()*

LAST_START_TIME_ACC (see section 4.12) specifies the accuracy of the returned *lastStartTime* value.

LAST_START_TIME_WCET (see section 4.15) specifies the *maximum processing time* for correct joint real-time operation of *radio application* and *transceiver*.

3.1.17.2.2 Declaration

The declaration of the operation **is specified as**:

```
void getLastStartTime(
    out TimeSpec lastStartTime,
    out BurstNumber lastBurstNumber);
```

3.1.17.2.3 Parameters

Name	Type	Description
lastStartTime	**TimeSpec** (see § 3.4.16)	Value of *transceiver time* for the start time of last created burst.
lastBurstNumber	**BurstNumber** (see § 3.4.5)	Number of the last created burst.

Table 21 Specification of *getLastStartTime()* parameters

3.1.17.2.4 Return value

None.

3.1.17.2.5 Originator

Radio application.

3.1.17.2.6 Exceptions

None.

3.1.17.2.7 Behavior requirements

An *active instance* of **`TimeAccess`** **shall**, on a call to *getLastStartTime()*, if no burst was created by the *channels* before the call to *getLastStartTime()*:

- Set the return value of *lastStartTime* to **`UndefinedTimeSpec`** (see section 3.4.16),
- Set the return value of *lastBurstNumber* to zero (`0`),
- Return the call to the *radio application*.

An *active instance* of **`TimeAccess`** **shall**, on a call to *getLastStartTime()*, if at least one burst was created by the *channels* before the call to *getLastStartTime()*:

- Set the return value of *lastStartTime* to a value belonging to the actual *start time* of the *last burst* created by the *channels* ± **`LAST_START_TIME_ACC`**,
- Set the return value of *lastBurstNumber* to the *burst number* of the *last burst* created by the *channels*,
- Return the call to the *radio application*.

3.1.18 Transceiver::Strobing::AppplicationStrobe

3.1.18.1 triggerStrobe Operation

3.1.18.1.1 Overview

triggerStrobe() provides the *channel* with a strobe occurrence.

3.1.18.1.2 Associated properties

`TRIGGER_STROBE_WCET` (see section 4.15) specifies the worst-case execution time of the primitive.

3.1.18.1.3 Declaration

The declaration of the operation **is specified as**:

```
void triggerStrobe( void);
```

3.1.18.1.4 Parameters

None.

3.1.18.1.5 Returned value

None.

3.1.18.1.6 Originator

Radio application.

3.1.18.1.7 Exceptions

None.

3.1.18.1.8 Behavior requirements

A *channel* **shall**, on a call to *triggerStrobe()*:

- Register the triggered strobe as a strobe occurrence for the **ApplicationStrobe** strobe source,
- Return the call to the *radio application*.

3.2 Exceptions

3.2.1 Specification

An *exception* **is defined as** an abnormal situation related to the calling context or to parameters values, detected during execution of a called *operation* of a *provide service* (see section 2.1).

General exceptions **are specified by** the following table:

Name	Description	See §
NoAlternateReferencing	Applies to: an *active instance* of RelativeCreation in a *simplex transceiver* or in a *duplex transceiver* with ALTERNATE_REFERENCING equal to false. Condition: the value of *requestCrossReference* in a call to *createRelativeBurst()* is equal to true while the *transceiver instance* is *simplex* or ALTERNATE_REFERENCING is false.	3.1.4 4.7
NoOngoingProcessing	Applies to: *active instance* of TerminationContol or Retuning. Condition: *setBlockLength()* or *retune()* is called while the *channels* are not in PROCESSING state.	3.1.7 3.1.12 2.3.1
StrobeSource	Applies to: an *active instance* of StrobedCreation. Condition: the value of *requestedStrobeSource* in a call to *createStrobedBurst()* has a corresponding field in STROBE_SOURCES equal to false.	3.1.6 4.7

Table 22 Specification of general exceptions

Range exceptions **are specified by** the following table:

Name	Description	See §
MinBlockLength MaxBlockLength	Applies to: an *active instance* of a *burst creation service* or Termination. Condition: the value of *requestedLength* in call to a *creation operation* or *setBlockLength()* is not equal to UndefinedBlockLength and is lower / greater than MIN_BLOCK_LENGTH / MAX_BLOCK_LENGTH.	3.1.7 4.7
MinCarrierFreq MaxCarrierFreq	Applies to: an *active instance* of InitialTuning or Retuning. Condition: the value of *requestedFrequency* in a call to *retune()* or *setTuning()* is not equal to UndefinedCarrierFreq and is lower / greater than MIN_CARRIER_FREQ / MAX_CARRIER_FREQ.	3.1.11 3.1.12 4.7
MinFromOngoing MaxFromOngoing	Applies to: an *active instance* of Retuning. Condition: the value of *requestedDelay* in a call to *retune()* is not equal to UndefinedDelay and is lower / greater than MIN_FROM_ONGOING / MAX_FROM_ONGOING.	3.1.12 4.7
MinFromPrevious MaxFromPrevious	Applies to: an *active instance* of RelativeCreation. Condition: the value of *requestedDelay* in a call to *createRelativeBurst()* is lower / greater than MIN_FROM_PREVIOUS / MAX_FROM_PREVIOUS.	3.1.4 4.7
MinFromStrobe MaxFromStrobe	Applies to: an *active instance* of StrobedCreation. Condition: the value of *requestedDelay* in a call to *createStrobedBurst()* is lower / greater than MIN_FROM_STROBE / MAX_FROM_STROBE.	3.1.6 4.7
MinGain MaxGain	Applies to: an *active instance* of InitialTuning or Retuning. Condition: the value of *requestedGain* in a call to *retune()* or *setTuning()* is not equal to UndefinedGain and is lower / greater than MIN_GAIN / MAX_GAIN.	3.1.11 3.1.12 4.7
MaxNanoseconds	Applies to: an *active instance* of AbsoluteCreation. Condition: the value of field *nanoseconds* of *requestedStartTime* in a call to *createAbsluteBurst()* is greater than 999.999.999.	3.1.5
MaxRxPacketsLength	Applies to: an *active instance* of RxPacketsLengthControl. Condition: the value of *requestedLength* in a call to *setRxPacketsLength()* is greater than MAX_PACKETS_LENGTH.	3.1.10 4.7
MaxTuningPreset	Applies to: an *active instance* of InitialTuning. Condition: the value of *requestedPreset* in a call to *setTuning()* is greater than MAX_TUNING_PRESET.	3.1.11 4.7
MaxTxPacketsLength	Applies to: an *active instance* of SamplesTransmission. Condition: the length of *txPacket* in a call to *pushTxPacket()* is greater than MAX_PACKETS_LENGTH.	3.1.8 4.7

Table 23 Specification of range exceptions

MILT exceptions **are specified by** the following table:

Name	Description	See §
AbsoluteMILT	Applies to: an *active instance* of **AbsoluteCreation**. Condition: the invocation time of *scheduleAbsoluteBurst()* does not respect **ABSOLUTE_MILT**.	3.1.5 4.13
RelativeMILT	Applies to: an *active instance* of **RelativeCreation**. Condition: the invocation time of *scheduleRelativeBurst()* does not respect **RELATIVE_MILT**.	3.1.4 4.13
RetuningMILT	Applies to: an *active instance* of **Retuning**. Condition: the invocation time *retune()* does not respect **RETUNING_MILT**.	3.1.12 4.6
TuningMILT	Applies to: an *active instance* of **InitialTuning**. Condition: the invocation time of *setTuning()* does not respect **TUNING_MILT**.	3.1.11 4.3.3
TxPacketsMILT	Applies to: an *active instance* of **SamplesTransmission**. Condition: the invocation time of *pushTxPacket()* does not respect **TX_PACKET_MILT**.	3.1.9 4.6

Table 24 Specification of MILT exceptions

3.2.2 Associated properties

EXCEPTIONS_SUPPORT (see section 4.4) specifies if exceptions are supported.

EXCEPTIONS (see section 4.4) specifies for each *exception*, if **EXCEPTIONS_SUPPORT** is equal to **true**, how any active instance of a *provide service* behave when the *exception* occurs:

- Reaction of the *provide service*,
- Need to raise the *exception*.

3.2.3 Behavior requirements

The *applicative handler* of an *exception* **<exceptionName>** **is defined as** a part of the *radio application* dedicated to handling of **<exceptionName>** occurrences.

The *exception raising* of an *exception* **<exceptionName>** **is defined as** branching the execution of the *radio application* to an *applicative handler* of **<exceptionName>** instead of waiting for the called operation to return.

The applied *PSM* (see section 1.1) specifies how *exception raising* is realized.

An *active instance* of a *provide service*, when **<exceptionName>** occurs and **EXCEPTIONS.<exceptionName>.reaction** is equal to **fatal**, has *unspecified* behavior.

An *active instance* of a *provide service* **shall**, when **<exceptionName>** occurs and **EXCEPTIONS.<exceptionName>.reaction** is equal to **resetting**:

- Trigger a RuntimeReset transition,
- If EXCEPTIONS.<exceptionName>.isNotified is equal to true, perform *exception raising*.

An *active instance* of a *provide service* **shall**, when <exceptionName> occurs and EXCEPTIONS.<exceptionName>.reaction is equal to callIgnoring:

- Implement no requirement of the nominal execution of the called operation,
- If EXCEPTIONS.<exceptionName>.isNotified is equal to true, perform *exception raising*.

3.3 Attributes

This section specifies *channels attributes* referenced by the remainder of the specification.

All channel attributes are virtual: *transceiver instances* are not required to make them accessible to *radio applications*.

3.3.1 Channels attributes

The *initial value* of a *channels* attribute **is defined as** the value of an attribute when *channels* enter the OPERATING state (see section 2.3.1).

3.3.1.1 burstCount

burstCount attribute **is specified as** the number of *bursts* created since the last entry in the OPERATING state (see section 2.3).

The associated declaration **is specified as**:

```
BurstNumber burstCount;
```

The *initial value* of burstCount **is specified as** 0 (zero).

Value of burstCount is incremented during INITIATING state of CreationControl, as specified in section 2.3.2.

3.3.1.2 applicableRxPacketsLength

applicableRxPacketsLength attribute **is specified as** the length of the *Rx packets* sent by an *Rx channel* with *pushRxPacket()* (see section 3.1.7).

The associated declaration **is specified as**:

```
PacketLength applicableRxPacketsLength;
```

INIT_RX_PACKETS_LENGTH (see section 4.6) specifies the *initial value* of applicableRxPacketsLength.

Value of applicableRxPacketsLength is changed by *radio applications* using *setRxPacketsLength()* (see section 3.1.10).

3.3.2 Processing attributes

3.3.2.1 applicableTuningPreset

applicableTuningPreset attribute **is specified as** a reference to the *transmit transfer function* (see section 1.2.4) or the *receive transfer function* (see section 1.2.5) applied by *channels* during **PROCESSING** state (see section 2.3).

The associated declaration **is specified as**:

```
TuningPreset applicableTuningPreset;
```

applicableTuningPreset ranges from **1** (one) to **MAX_TUNING_PRESET** (see section 4.7).

For *channels* with no active instance of **InitialTuning**, the value of **applicableTuningPreset** is equal to **1** and cannot be modified.

For *channels* with an active instance of **InitialTuning**, no *initial value* of **applicableTuningPreset** is specified.

Value of **applicableTuningPreset** is controlled by *radio applications* using *setTuning()* (see section 3.1.11).

3.3.2.2 applicableCarrierFreq

applicableCarrierFreq attribute **is specified as** the *carrier frequency* (see section 1.2.2.2) applied by *channels* during **PROCESSING** state (see section 2.3).

The associated declaration **is specified as**:

```
CarrierFreq applicableCarrierFreq;
```

applicableCarrierFreq ranges from **MIN_CARRIER_FREQ** to **MAX_CARRIER_FREQ** (see section 4.7).

For *channels* with no active instance of **InitialTuning**, **INIT_CARRIER_FREQ** (see section 4.6) specifies the value of **applicableCarrierFreq** at beginning of the first *burst*.

For *channels* with an active instance of **InitialTuning**, no *initial value* of **applicableCarrierFreq** is specified.

Value of **applicableCarrierFreq** is controlled by *radio applications* using *setTuning()* (see section 3.1.11) and *retune()* (see section 3.1.12).

3.3.2.3 applicableGain

applicableGain attribute **is specified as** the *transmit gain* (see section 1.2.4.4) or the *receive gain* (see section 1.2.5) applied by *channels* during **PROCESSING** state (see section 2.3).

The associated declaration **is specified as**:

```
Gain applicableGain;
```

applicableGain ranges from MIN_GAIN to MAX_GAIN (see section 4.7).

For *channels* with no active instance of InitialTuning, INIT_GAIN (see section 4.6) specifies the value of applicableGain at beginning of the first *burst*.

For *channels* with an active instance of InitialTuning, no *initial value* of applicableGain is specified.

Value of applicableGain is controlled by *radio applications* using *setTuning()* (see section 3.1.11) and *retune()* (see section 3.1.12).

3.3.2.4 applicableLength

applicableLength attribute **is specified as** the length of the *baseband block* to be processed by *channels* during PROCESSING state (see section 2.3).

The associated declaration **is specified as**:

```
BlockLength applicableLength;
```

Undefined applicableLength is equal to UndefinedBlockLength (see section 3.4.3).

Defined applicableLength ranges from MIN_BLOCK_LENGTH to MAX_BLOCK_LENGTH (see section 4.7).

No *initial value* of applicableLength is specified.

Value of applicableLength is controlled by *radio applications* using *creation operations* (see section 2.4.2) and *setBlockLength()* (see section 3.1.7).

3.3.2.5 sampleCount

sampleCount attribute **is specified as** the number of *samples* of the *baseband block* processed by *channels* since entry in the PROCESSING state (see section 2.3).

The associated declaration **is specified as**:

```
SampleNumber sampleCount;
```

The *start value* of sampleCount **is specified as** 1 (one) for the *first sample* of the *baseband block*.

Value of sampleCount is incremented during PROCESSING state of Channels, as specified in section 2.3.1.

3.4 Types

3.4.1 Base assumptions

The IDL keywords used for specification of types are:

- For Basic Types:
  - 16-bit integers: ***short***, ***unsigned short***,
  - 32-bit integers: ***long***, ***unsigned long***,
  - 64-bit integers: ***long long***, ***unsigned long long***,
  - Others: ***float***, ***boolean***,
- For Constructed Types: ***typedef***, ***struct***, ***enum***,
- For Template Types: ***sequence***.

This makes the specification compliant with the Full Profile or [Ref5], and with the ULw Profile augmented by ***long long*** and ***float*** basic types.

3.4.2 BasebandPacket

BasebandPacket type **is specified as** a sequence of *baseband samples*.

The associated declaration **is specified as**:

```
typedef sequence <BasebandSample> BasebandPacket;
```

BasebandPacket is used by *pushRxPacket()* (see section 3.1.7) and *pushTxPacket()* (see section 3.1.9).

3.4.3 BlockLength

BlockLength type **is specified as** a 32-bit unsigned integer number of *baseband samples* to be processed by *Tx channels* or *Rx channels* during a *processing phase*.

UndefinedBlockLength **is specified as** the reserved value specifying an *undefined* value of ***BlockLength***.

The associated declarations **are specified as**:

```
typedef unsigned long BlockLength;
const BlockLength UndefinedBlockLength = 0xFFFFFF;
```

BlockLength is used by *startBurst()* (see section 3.1.3), *scheduleRelativeBurst()* (see section 3.1.4), *scheduleAbsoluteBurst()* (see section 3.1.5) and *scheduleStrobedBurst()* (see section 3.1.6).

3.4.4 BasebandSample

`BasebandSample` type **is specified as** the structure representing *baseband samples*, with field `valueI` for the *in-phase component* and field `valueQ` for the *quadrature component* (see section 1.2.2.1).

The associated declaration **is specified as**:

```
struct BasebandSample {IQ valueI, IQ valueQ};
```

`BasebandSample` is used by declaration of `IQ` type (see section 3.4.11).

3.4.5 BurstNumber

`BurstNumber` type **is specified as** a 32-bit unsigned integer that specifies a burst number.

The associated declaration **is specified as**:

```
typedef unsigned long BurstNumber;
```

`BurstNumber` is used by *setTuning()* (see section 3.1.11) and `burstCount` attribute (see 3.3.1.1).

3.4.6 CarrierFreq

`CarrierFreq` type **is specified as** an unsigned integer that specifies a *carrier frequency* (f_c).

`CARRIER_FREQ_TYPE` (see section 4.3) specifies if `CarrierFreq` is 32-bit or 64-bit.

A `CarrierFreq` value is expressed in hertz (Hz).

`UndefinedCarrierFreq` **is specified as** the reserved value specifying an *undefined* value of `CarrierFreq`.

The associated declarations **are specified as**, if `CARRIER_FREQ_TYPE` is equal to *32bit*:

```
typedef unsigned long CarrierFreq; // in Hz
const CarrierFreq UndefinedCarrierFreq = 0xFFFFFFFF;
```

The associated declarations **are specified as**, if `CARRIER_FREQ_TYPE` is equal to *64bit*:

```
typedef unsigned long long CarrierFreq; // in Hz
const CarrierFreq UndefinedCarrierFreq = 0xFFFFFFFFFFFFFFFF;
```

`CarrierFreq` is used by *setTuning()* (see section 3.1.11) and retune*()* (see section 3.1.12).

3.4.7 Delay

`Delay` type **is specified as** an unsigned integer that specifies a delay from the *start time* of an ongoing *processing phase*.

`DELAY_TYPE` (see section 4.3) specifies if `Delay` is 32-bit or 64-bit.

A `Delay` value is expressed in nanoseconds (ns).

UndefinedDelay **is specified as** the reserved value specifying an *undefined* value of ***Delay***.

The associated declarations **are specified as**, if **DELAY_TYPE** is equal to *32bit*:

```
typedef unsigned long Delay; // in ns
const Delay UndefinedDelay = 0xFFFFFFFF;
```

The associated declarations **are specified as**, if **DELAY_TYPE** is equal to *64bit*:

```
typedef unsigned long long Delay; // in ns
const Delay UndefinedDelay = 0xFFFFFFFFFFFFFFFF;
```

Delay is used by *scheduleRelativeBurst()* (see section 3.1.4), *scheduleStrobedBurst()* (see section 3.1.6) and *retune()* (see section 3.1.12).

3.4.8 Error

Error type **is specified as** an enumeration identifying an *error*.

The associated declaration **is specified as**:

```
enum Error {
    errorDelayedTuning,
    errorTuningTimeout,
    errorDelayedFirstSample,
    errorFirstSampleTimeout,
    errorTransmissionUnderflow,
    errorReceptionOverflow,
    errorShorterTransmittedBlock,
    errorLongerTransmittedBlock};
```

Error is used by *notifyError()* (see section 3.1.14.1).

3.4.9 Event

Event type **is specified as** an enumeration identifying an *event*.

The associated declaration **is specified as**:

```
enum Event {
    eventProcessingStart,
    eventProcessingStop,
    eventSilenceStart,
    eventSilenceStop};
```

Event is used by *notifyEvent()* (see section 3.1.13.1).

3.4.10 Gain

Gain type **is specified as** a signed 16-bit integer that specifies a gain (*G*).

A ***Gain*** value is expressed in tenths of decibels (1/10 dB).

UndefinedGain **is specified as** the reserved value specifying an *undefined* value of ***Gain***.

The associated declarations **are specified as**:

```
typedef short Gain; // in 1/10 dB
const Gain UndefinedGain = 0xFFFF;
```

`Gain` is used by *setTuning()* (see section 3.1.11) and retune*()* (see section 3.1.12).

3.4.11 IQ

`IQ` type **is specified as** the representation of I (in-phase) and Q (quadrature) components of a *baseband sample*.

`IQ_TYPE` (see section 4.3) specifies if **`IQ`** type is 16-bit, 32-bit or floating point.

Integer values of ***`IQ`*** **shall** be signed 2-complement MSB-aligned.

The declaration of *IQ* **is specified as**, if **`IQ_TYPE`** is equal to *16bit*:

```
typedef short IQ;
```

The declaration of *IQ* **is specified as**, if **`IQ_TYPE`** is equal to *32bit*:

```
typedef long IQ;
```

The declaration of *IQ* **is specified as**, if **`IQ_TYPE`** is equal to *floatingPoint*,

```
typdef float IQ;
```

`IQ` is used for declaration of ***`BasebandSample`*** type (see section 3.4.4).

3.4.12 MetaData

`TxMetaData` and ***`RxMetaData`*** types **are specified as** structures of *unspecified* fields optionally used to attach meta-data to transferred *baseband packets*.

The associated declarations are user-defined, and **shall** be specified as follows:

```
typedef struct TxMetaData {
    <user-defined>};

typedef struct RxMetaData {
    <user-defined>};
```

`TxMetaData` is used by *pushTxPacket()* (see section 3.1.9) and ***`RxMetaData`*** is used by *pushRxPacket()* (see section 3.1.8).

3.4.13 PacketLength

`PacketLength` type **is specified as** a 32-bit unsigned integer that identifies the length of a *packet*.

The associated declarations **are specified as**:

```
typedef unsigned long PacketLength;
```

`PacketLength` is used by *setRxPacketsLength()* (see section 3.1.10) and **`applicableRxPacketsLength`** (see section 3.3.1.2).

3.4.14 SampleNumber

`SampleNumber` type **is specified as** a 32-bit unsigned integer that specifies a sample number.

The associated declaration **is specified as**:

```
typedef unsigned long SampleNumber;
```

`SampleNumber` is used by *indicateGain()* (see section 3.1.15.1) and **`sampleCount`** attribute (see 3.3.1.2).

3.4.15 StrobeSource

`StrobeSource` type **is specified as** an enumeration that specifies the referenced strobe source for strobed creation of a burst, as specified in section 3.1.6.

The associated declaration **is specified as**:

```
enum StrobeSource {
    ApplicationStrobe,
    TimeRef_PPS,
    GNSS_PPS,
    UserStrobe1,
    UserStrobe2,
    UserStrobe3,
    UserStrobe4};
```

`StrobeSource` is used by *scheduleStrobedBurst()* (see section 3.1.6).

3.4.16 TimeSpec

`TimeSpec` type **is specified as** a structure that specifies a value of *transceiver time*, composed of 32-bit unsigned integer fields for seconds and nanoseconds.

The *seconds* field value is expressed in seconds (s).

The *nanoseconds* field value is expressed in nanoseconds (ns).

`UndefinedTimeSpec` **is specified as** the reserved value specifying an *undefined* value of **`TimeSpec`**.

The associated declarations **are specified as**:

```
struct TimeSpec {
    unsigned long seconds,            // in seconds
    unsigned long nanoseconds};       // in nanoseconds (<1.000.000.000)
const TimeSpec UndefinedTimeSpec = {0xFFFFFFFF, 0xFFFFFFFF};
```

`TimeSpec` is used by *scheduleAbsoluteBurst()* (see section 3.1.5), *getCurrentTime()* and *getLastStartTime()* (see section 3.1.15).

3.4.17 TuningPreset

TuningPreset type **is specified as** a 16-bit unsigned integer that identifies a tuning preset.

UndefinedTuningPreset **is specified as** the reserved value specifying an *undefined* value of ***TuningPreset***.

The associated declarations **are specified as**:

```
typedef unsigned short TuningPreset;
const TuningPreset UndefinedTuningPreset = 0xFFFF;
```

TuningPreset is used by *setTuning()* (see section 3.1.11).

4 Properties

This section specifies the *Transceiver Properties*, which characterize a *transceiver instance*, once it has been reconfigured in accordance to needs of the supported *radio application*.

4.1 Introduction

4.1.1 Properties

A *property* **is defined as** an attribute of a *transceiver instance* which value is defined when the *channels* have reached the CONFIGURED state.

The value of a *property* cannot be modified until the *channels* have exited the CONFIGURED state.

Note: future versions of the *specification* may enable modification of *property* values.

The remainder of the section specifies *properties* and their *base name*, also denoted <BaseName>.

Depending on cases, a unique *property* can fully characterize a *transceiver instance*, or multiple *properties* can be required.

4.1.2 Properties naming

The name of a unique *property* **shall** be the <BaseName> of the *property*.

The names of multiple *properties* are constructed from the <BaseName> of the *property* with usage of prefixes or postfixes.

The name of multiple *properties* that differ between *Tx channels* and *Rx channels* **shall** be constructed with TX_ and RX_ prefixes added before the <BaseName>.

A *property* which *base name* starts with TX_ (resp. RX_) only applies to *Tx channels* (resp. *Rx channels*).

The name of multiple *properties* that differ according to *conditions* **shall** be constructed with the condition-dependent <Condition> postfixes added after the <BaseName> and a separation composed of two (2) underscores (__).

Any *conditions* and associated <Condition> postfixes can be *user-defined*.

For *rapidity properties*, section 4.8 specifies standard *conditions* and <Condition> postfixes.

4.1.3 Portability engineering support

The *configuration expectations* of a *radio application* **are defined as** the properties values of each used *transceiver instance* required for correct operation after the CONFIGURED state is reached.

The *configuration capabilities* of a *transceiver implementation* **are defined as** the properties values possibly supported by the *transceiver* after the CONFIGURED state is reached.

Porting feasibility can be evaluated through comparison of the *radio application*'s *configuration expectations* with *transceiver*'s *configuration capabilities*.

Note: derived specifications may standardize machine readable meta-data for expression of *configuration expectations*, enabling automation of porting feasibility evaluations and, for some advanced implementations, of the configuration of the *transceiver instances*.

4.1.4 Profiles

A *profile* of the *specification* **is defined as** a standard that specifies values of *properties* for *radio applications* and *transceivers* to facilitate or even guarantee that porting of any compliant *radio application* is feasible on any compliant *transceiver implementation*.

Note: development of *profiles* is out of the scope of the *specification*, but may be standardized by derived specifications.

4.2 Structure

A *structure property* **is defined as** a *property* that specifies an aspect related to the structure of a *transceiver instance*.

Structure properties **are specified by** the following table:

Base name	Type	Description	See §
TX_CHANNELS	*unsigned short*	Applies to: any *transceiver instance*. Specifies: number of *Tx channels* (equal to number of *active instances* of SamplesTransmission).	1.2.1
RX_CHANNELS	*unsigned short*	Applies to: any *transceiver instance*. Specifies: number of *Rx channels* (equal to number of *active instances* of SamplesReception).	1.2.1
DUPLEX	Enumeration (see below)	Applies to: a *duplex transceiver* (TX_CHANNELS > 0 and RX_CHANNELS > 0). Specifies: duplex type of the *transceiver instance*: ▪ fullDuplex, ▪ halfDuplex.	1.2.1
TX_SHAPING	Enumeration (see below)	Applies to: *Tx channels*. Specifies: shaping of *Tx bursts*: ▪ nominal, ▪ specific.	1.2.4
TX_SERVICES	*ActiveServices* (see below)	Applies to: *Tx channels*. Specifies: for each *service* except SamplesTransmission, if one *active instance* is attached to *Tx channels*.	1.3.3
RX_SERVICES	*ActiveServices* (see below)	Applies to: *Rx channels*. Specifies: for each *service* except SamplesReception, if one *active instance* is attached to *Rx channels*.	1.3.3
TIME_COUPLING	Enumeration (see below)	Applies to: *channels* with *active instance* of AbsoluteCreation. Specifies: coupling of *transceiver time*: ▪ autonomous: uncorrelated with any other time, ▪ coupled: identical to another time, ▪ coupledToTerminalTime: identical to *Terminal Time* of *Timing Service API* (see [Ref7]).	3.1.5

Table 25 Structure properties

The declaration of DUPLEX **is specified as**:

```
enum DUPLEX {fullDuplex, halfDuplex};
```

The declaration of TX_SHAPING **is specified as**:

```
enum TX_SHAPING {nominal, specific};
```

The declarations for **TX_SERVICES** and **RX_SERVICES are specified as**:

```
typedef boolean isActive;

typedef struct {
    // Provide services
    isActive reset,
    isActive radioSilence,
    isActive directCreation,
    isActive relativeCreation,
    isActive absoluteCreation,
    isActive strobedCreation,
    isActive termination,
    isActive rxPacketsLengthControl,
    isActive initialTuning,
    isActive retuning,
    isActive gainLocking,
    isActive timeAccess,
    isActive applicationStrobe,

    // Use services
    isActive events,
    isActive errors
    isActive gainChanges,
} ActiveServices;

ActiveServices TX_SERVICES;
ActiveServices RX_SERVICES;
```

The following consistency conditions apply to fields of **TX_SERVICES** and **RX_SERVICES**:

- At least one among **directCreation**, **relativeCreation**, **absoluteCreation** and **strobedCreation** is equal to **true**,
- **rxPacketsLengthControl** of **TX_SERVICES** is equal to **false**,
- **timeAccess** is equal to **false** if **relativeCreation** is equal to **false**,
- **applicationStrobe** is equal to **false** if **strobedCreation** is equal to **false**.

The declaration of **TIME_COUPLING is specified as**:

```
enum TIME_COUPLING {autonomous, coupled, coupledToTerminalTime};
```

4.3 Behavior

A *behavior property* **is defined as** a *property* that specifies an aspect relative to the behavior of a *transceiver instance*.

Behavior properties **are specified by** the following table:

Base name	Type	Description	See §
TUNING_ASSOCIATION	Enumeration (see below)	Applies to: *channels* with an *active instance* of InitialTuning. Specifies: search condition among stored *tuning parameters sets* applicable during INITIATING: ▪ sequential, ▪ burstReferencing.	2.3.2
AGC	Enumeration (see below)	Applies to: *Rx channels*. Specifies: nature of the implemented *AGC*: ▪ noAGC, ▪ earlyControl, ▪ permanentControl.	2.3.1
ALC	Enumeration (see below)	Applies to: *Tx channels*. Specifies: nature of the implemented *ALC*: ▪ noALC, ▪ activeALC.	2.3.1
TUNING_TIMEOUT	*unsigned long*	Applies to: *channels* with an *active instance* of InitialTuning, if ERRORS.errTuningDelayed.reaction is equal to mitigating. Specifies: timeout value, in nanoseconds (ns), for triggering of errorTuningTimeout.	3.1.14
1ST_SAMPLE_TIMEOUT	*unsigned long*	Applies to: *Tx channels* with at least one *active instance* of *timely creation services*, if ERRORS.err1stSampleDelayed.reaction is equal to mitigating. Specifies: timeout value, in nanoseconds (ns), for triggering of error1stSampleTimeout.	3.1.14

Table 26 Behavior properties

The declaration of TUNING_ASSOCIATION **is specified as**:

```
enum TUNING_ASSOCIATION {sequential, burstReferencing};
```

The declaration of AGC **is specified as**:

```
enum AGC {noAGC, startupAGC, permanentAGC};
```

The declaration of ALC **is specified as**:

```
enum ALC {noALC, activeALC};
```

4.4 Notifications

A *notification property* **is defined as** a *property* that specifies an aspect relative to notifications made by a *transceiver instance* to the *radio application*.

Notification properties **are specified by** the following table:

Base name	Type	Description	See §
EXCEPTIONS_SUPPORT	*boolean*	Applies to: all *channels*. Specifies: if exceptions are supported.	3.2
EXCEPTIONS	Structure (see below)	Applies to: all *channels*. Specifies: an **exceptionHandling** field for each standard *exception*, which specifies the *reaction* to occurrences of the *exception* and if the *exception* is raised to the *radio application* with the exception notification mechanism.	3.2
EVENTS	Structure (see below)	Applies to: *channels* with an *active instance* of **Events**. Specifies: an **isNotified** field for each *event*, which specifies if occurrences are notified to the *radio application* with *notifyEvent()*.	3.1.13
ERRORS	Structure (see below)	Applies to: *channels* with an *active instance* of **Errors**. Specifies: an **errorHandling** field for each *error*, which specifies the *reaction* to occurrences of the *error* and if occurrences are notified to the *radio application* with *notifyError()*.	3.1.14

Table 27 Notification properties

The declarations for **ERRORS** **are specified as**:

```
typedef struct{
    enum reaction {fatal, reset, mitigation},
    boolean isNotified}
errorHandling;

struct ERRORS {
    errorHandling error1stSampleDelayed,
    errorHandling error1stSampleTimeout,
    errorHandling errorBurstOverlap,
    errorHandling errorRxOverflow,
    errorHandling errorShorterTxBlock,
    errorHandling errorTxUnderflow,
    errorHandling errorTuningDelayed,
    errorHandling errorTuningTimeout};
```


The declarations for **EXCEPTIONS** **are specified as**:

```
typedef struct{
    enum reaction {fatal, resetting, callIgnoring}}
    boolean isRaised}
exceptionHandling;

struct EXCEPTIONS {
    // General exceptions
    exceptionHandling NoAlternateReferencing,
    exceptionHandling NoOngoingProcessing,
    exceptionHandling StrobeSource,

    // Range exceptions
    exceptionHandling MaxBlockLength,
    exceptionHandling MinBlockLength,
    exceptionHandling MaxCarrierFreq,
    exceptionHandling MinCarrierFreq,
    exceptionHandling MaxFromOngoing,
    exceptionHandling MinFromOngoing,
    exceptionHandling MinFromPrevious,
    exceptionHandling MaxFromPrevious,
    exceptionHandling MaxFromStrobe,
    exceptionHandling MinFromStrobe,
    exceptionHandling MaxGain,
    exceptionHandling MinGain,
    exceptionHandling MaxNanoseconds,
    exceptionHandling MaxRxPacketsLength,
    exceptionHandling MaxTuningPreset,
    exceptionHandling MaxTxPacketsLength

    // MILT exceptions
    exceptionHandling AbsoluteMILT,
    exceptionHandling RelativeMILT,
    exceptionHandling RetuningMILT,
    exceptionHandling TuningMILT,
    exceptionHandling TxPacketsMILT};
```

The declarations for **EVENTS** **are specified as**:

```
typedef boolean isNotified;

struct EVENTS {
    isNotified eventProcessingStart,
    isNotified eventProcessingStop,
    isNotified eventSilenceStart,
    isNotified eventSilenceStop};
```

4.5 Interface declaration

An *interface declaration property* **is defined as** a *property* that specifies an aspect relative to the declaration of a *service interface*.

Interface declaration properties **are specified by** the following table:

Base name	Type	Description	See §
CARRIER_FREQ_TYPE	Enumeration (see below)	Applies to: CarrierFreq type. Specifies: type used (32-bit or 64-bit).	3.4.6
DELAY_TYPE	Enumeration (see below)	Applies to: Delay type. Specifies: type used (32-bit or 64-bit).	3.4.7
IQ_TYPE	Enumeration (see below)	Applies to: IQ type. Specifies: type used (16-bit, 32-bit or floating point).	3.4.11
TX_META_DATA	boolean	Specifies if user-defined meta-data are attached to the *Tx packets* forwarded to *Tx channels*.	3.1.9
RX_META_DATA	boolean	Specifies if user-defined meta-data are attached to the *Rx packets* obtained from *Rx channels*.	3.1.8

Table 28 Interface declaration properties

The associated declarations **are specified as**:

```
enum CARRIER_FREQ_TYPE {int32, int64};
enum DELAY_TYPE {int32, int64};
enum IQ_TYPE {int16, int32, float32};
```

4.6 Initialization

An *initialization property* **is defined as** a *property* that specifies the conditions to be met by a *transceiver instance* when the CONFIGURED state is reached by its *Tx channels* and *Rx channels*.

Initialization properties **are specified by** the following table:

Base name	Type	Description	See §
INIT_RX_PACKETS_LENGTH	PacketLength (see § 3.4.12)	Applies to: all *Rx channels*. Specifies: *initial value* of applicableRxPacketsLength.	3.3.1
INIT_CARRIER_FREQ	CarrierFreq (see § 3.4.6)	Applies to: *channels* with no active instance of InitialTuning. Specifies: the value of applicableCarrierFreq at beginning of the first *burst*.	3.3.2
INIT_GAIN	Gain (see § 3.4.10)	Applies to: *channels* with no active instance of InitialTuning. Specifies: the value of applicableGain at beginning of first *burst*.	3.3.2

Table 29 Initialization properties

4.7 Parameters validity

A *parameter validity property* **is defined as** a *property* that specifies the validity conditions applicable to a *parameter* of a *primitive* of a *service interface*.

Parameters validity properties **are specified by** the following table:

Base name	Type	Description	See §
MIN_BLOCK_LENGTH MAX_BLOCK_LENGTH	*BlockLength* (see § 3.4.3)	Applies to: *requestedLength* not equal to UndefinedBlockLength in a call to a *creation operation*. Specifies: minimum and maximum value.	3.1.3 3.1.4 3.1.5 3.1.6 3.1.7
ALTERNATE_REFERENCING	*boolean*	Applies to: *requestedAlternate* in a call to *scheduleRelativeBurst()*. Specifies: if true value is supported.	3.1.4
MIN_FROM_PREVIOUS MAX_FROM_PREVIOUS	*Delay* (see § 3.4.7)	Applies to: *requestedDelay* in a call to *scheduleRelativeBurst()*. Specifies: minimum and maximum value.	3.1.4
STROBE_SOURCES	Structure (see below)	Applies to: *requestedStrobeSource* in *scheduleStrobedBurst()*. Specifies: for each *boolean* field attached to a *strobe source*, if the corresponding value of *requestedStrobeSource* is supported.	3.1.6
MIN_FROM_STROBE MAX_FROM_STROBE	*Delay* (see § 3.4.7)	Applies to: *requestedDelay* in a call to *scheduleStrobedBurst()*. Specifies: minimum and maximum value.	3.1.6
MAX_PACKETS_LENGTH	*PacketLength* (see § 3.4.12)	Applies to: length of *txPacket* in a call to *pushTxPacket()* or *requestedLength* in a call to *setRxPacketsLength()*. Specifies: maximum value. Note: minimum value is constant and equal to 1.	3.1.9 3.1.10
MAX_TUNING_PRESET	*TuningPreset* (see § 3.4.14)	Applies to: *requestedPreset* in a call to *setTuning()*. Specifies: maximum value. Note: minimum value is constant and equal to 1.	3.1.11
MIN_CARRIER_FREQ MAX_CARRIER_FREQ	*CarrierFreq* (see § 3.4.6)	Applies to: *requestedFrequency* not equal to UndefinedCarrierFreq in a call to *setTuning()* or *retune()*. Specifies: minimum and maximum value.	3.1.11 3.1.12
MIN_GAIN MAX_GAIN	*Gain* (see § 3.4.10)	Applies to: *requestedGain* not equal to UndefinedGain in a call to *setTuning()* or *retune()*. Specifies: minimum and maximum value.	3.1.11 3.1.12
MIN_FROM_ONGOING MAX_FROM_ONGOING	*Delay* (see § 3.4.7)	Applies to: *requestedDelay* in a call to *retune()*. Specifies: minimum and maximum value.	3.1.12

Table 30 Parameters validity properties

The declaration of STROBE_SOURCES **is specified as**:

```
typedef boolean isSupported;

struct STROBE_SOURCES {
    isSupported ApplicationStrobe,
    isSupported TimeRef_PPS,
    isSupported GNSS_PPS,
    isSupported UserStrobe1,
    isSupported UserStrobe2,
    isSupported UserStrobe3,
    isSupported UserStrobe4};
```

4.8 Rapidity

A *rapidity property* **is defined as** a property that specifies the rapidity of execution of a *transceiver instance*.

Rapidity properties **are specified as** indicated in the following table:

Base name	Type	Description	See §
INTER-PROCESSING	*unsigned long*	Applies to: *channels*. Specifies: minimum time, in nanoseconds (ns), between: ▪ *Termination time* of a burst (a StopProcessing transition), ▪ Activation time of the next burst (StartProcessing transition).	1.2.6
INTER-BURST	*unsigned long*	Applies to: *channels*. Specifies: minimum time, in nanoseconds (ns), between: ▪ *Stop time* of a burst (end of its *core burst*, at its *start time* plus *block length* / F_S^{BB}), ▪ *Start time* of the next burst (end of its *core burst*).	1.2.6
TUNING_DURATION	*unsigned long*	Applies to: *channels* with an *active instance* of Tuning. Specifies: maximum duration, in nanoseconds (ns), of the TUNING state.	2.3.1
RETUNING_DURATION	*unsigned long*	Applies to: *channels* with an *active instance* of Retuning. Specifies: maximum duration, in nanoseconds (ns), of the RETUNING state.	2.3.4
EARLY_AGC_DELAY	*unsigned long*	Applies to: *Rx channels* with AGC equal to earlyControl. Specifies: delay available after *start time* of a *Rx burst* for the *AGC* to have set the *receive gain*.	2.3.1

Table 31 Rapidity properties

Tuning conditions **are specified as** indicated in the following table:

<Condition> postfix	Condition
NO_TUNING_CHANGE	Applies to: INTER-BURST, INTER-PROCESSING and TUNING-DURATION of *channels* with an active instance of InitialTuning. Condition: the *applicable tuning parameters set* specifies no tuning change (*requestedTuningPreset* is equal to undefinedTuningPreset, *requestedCarrierFreq* is equal to UndefinedCarrierFreq and *requestedDelay* is equal to UndefinedDelay).
NEW_TUNING_PRESET	Applies to: INTER-BURST, INTER-PROCESSING and TUNING-DURATION of *channels* with an active instance of InitialTuning. Condition: the *applicable tuning parameters set* specifies a new *tuning preset* (*requestedTuningPreset* is **not** equal to undefinedTuningPreset).
NEW_FREQUENCY	Applies to: INTER-BURST, INTER-PROCESSING and TUNING-DURATION of *channels* with an active instance of InitialTuning and RETUNING_DURATION of *channels* with an active instance of Retuning. Condition: the *applicable tuning parameters set* specifies a new *frequency* with no *tuning preset* change (*requestedTuningPreset* is equal to undefinedTuningPreset and *requestedCarrierFreq* is **not** equal to UndefinedCarrierFreq).
NEW_GAIN	Applies to: INTER-BURST, INTER-PROCESSING and TUNING-DURATION of *channels* with an active instance of InitialTuning and RETUNING_DURATION of *channels* with an active instance of Retuning. Condition: the *applicable tuning parameters set* specifies a new *gain* with no other change (*requestedTuningPreset* is equal to undefinedTuningPreset, *requestedCarrierFreq* is equal to UndefinedCarrierFreq and *requestedDelay* is **not** equal to UndefinedDelay).

Table 32 Tuning conditions

See section 2.3.2.1.3 for further information regarding *applicable tuning parameters set*.

Duplex conditions **are specified as** indicated in the following table:

<Condition> postfix	Condition
TX-TX	Applicable to: INTER-BURST, INTER-PROCESSING and TUNING-DURATION of all *Tx channels*. Condition: the consecutive bursts are *Tx bursts*.
RX-RX	Applicable to: INTER-BURST, INTER-PROCESSING and TUNING-DURATION all *Rx channels*. Condition: the consecutive bursts are *Rx bursts*.
TX-RX	Applicable to: INTER-BURST, INTER-PROCESSING and TUNING-DURATION of half-duplex transceivers. Condition: the *previous burst* is a *Tx burst* and the next burst is a *Rx burst*.
RX-TX	Applicable to: INTER-BURST, INTER-PROCESSING and TUNING-DURATION half-duplex transceivers. Condition: the *previous burst* is a *Tx burst* and the next burst is a *Rx burst*.

Table 33 Duplex conditions

4.9 Storage

A *storage property* **is defined as** a *property* that specifies the number of calls to certain operations a *transceiver instance* can store before blocking further calls until storage is freed.

Storage properties **are specified by** the following table:

Base name	Type	Description	See §
CREATION_STORAGE	*unsigned short*	Applies to: all *Tx channels* and *Rx channels*. Specifies: maximum number of *creation operations* calls the *transceiver instance* can store.	3.1.3 3.1.4 3.1.5 3.1.6
TUNING_STORAGE	*unsigned short*	Applies to: *channels* with an *active instance* of InitialTuning. Specifies: maximum number of *setTuning()* calls the *transceiver instance* can store.	3.1.11
TX_BASEBAND_STORAGE	*unsigned long*	Applies to: *Tx channels*. Specifies: maximum number of *baseband samples* the *transceiver instance* can store for each *active instance* of SamplesTransmission.	3.1.9

Table 34 Storage properties

4.10 Levels

A *level property* **is defined as** a property that specifies the range of signal levels at the boundary of *channels*.

Level properties **are specified by** the following table:

Base name	Type	Description	See §
TX_MIN_BASEBAND_LEVEL TX_MAX_BASEBAND_LEVEL	*short*	Applies to: *Tx channels*. Specifies: minimum and maximum values of the level of *baseband signal* at input of *Tx channels*, in tenth of decibels relative to full scale (1/10 dBFS).	2.3.1
RX_MIN_RADIO_LEVEL RX_MAX_RADIO_LEVEL	*short*	Applies to: *Rx channels*. Specifies: minimum and maximum values of the level of *radio signal* at input of *Rx channels*, in tenth of decibels relative to one milliwatt (1/10 dBm).	2.3.1
RX_MIN_BASEBAND_LEVEL RX_MAX_BASEBAND_LEVEL	*short*	Applies to: *Rx channels*. Specifies: minimum and maximum values of the level of *baseband signal* at output of *Rx channels*, in tenth of decibels relative to full scale (1/10 dBFS).	2.3.1

Table 35 Level properties

4.11 Channelization

A *channelization property* **is defined as** a *property* that specifies each *tuning preset* supported by a *transceiver instance*.

Channelization properties **are specified by** the following table:

Base name	Type	Description	See §
CHANNEL_MASK	Structure (see below)	Applies to: all *tuning presets*. Specifies: the *channel mask* for the *transfer function*, to be respected during the PROCESSING state.	2.3.1
SAMPLING_FREQ_ACC	*unsigned long*	Applies to: *channels*. Specifies: accuracy of the *baseband sampling frequency*, in hertz (Hz), to be respected during the PROCESSING state.	2.3.1
CARRIER_FREQ_ACC	*CarrierFreq*	Applies to: *channels*. Specifies: accuracy of the *carrier frequency*, to be respected during the PROCESSING state.	2.3.1
GAIN_ACC	*Gain*	Applies to: *channels*. Specifies: accuracy of the *gain*, to be respected during the PROCESSING state.	2.3.1

Table 36 Channelization properties

One *property instance* of CHANNEL_MASK is specified for each value of tuning preset between 1 and MAX_TUNING_PRESET (see section 4.7).

The associated names **are specified as**:

- CHANNEL_MASK if MAX_TUNING_PRESET is equal to 1,
- CHANNEL_MASK_<PresetNumber> if MAX_TUNING_PRESET is greater than 1.

The fields of *channel masks* **are specified by** the following figure:

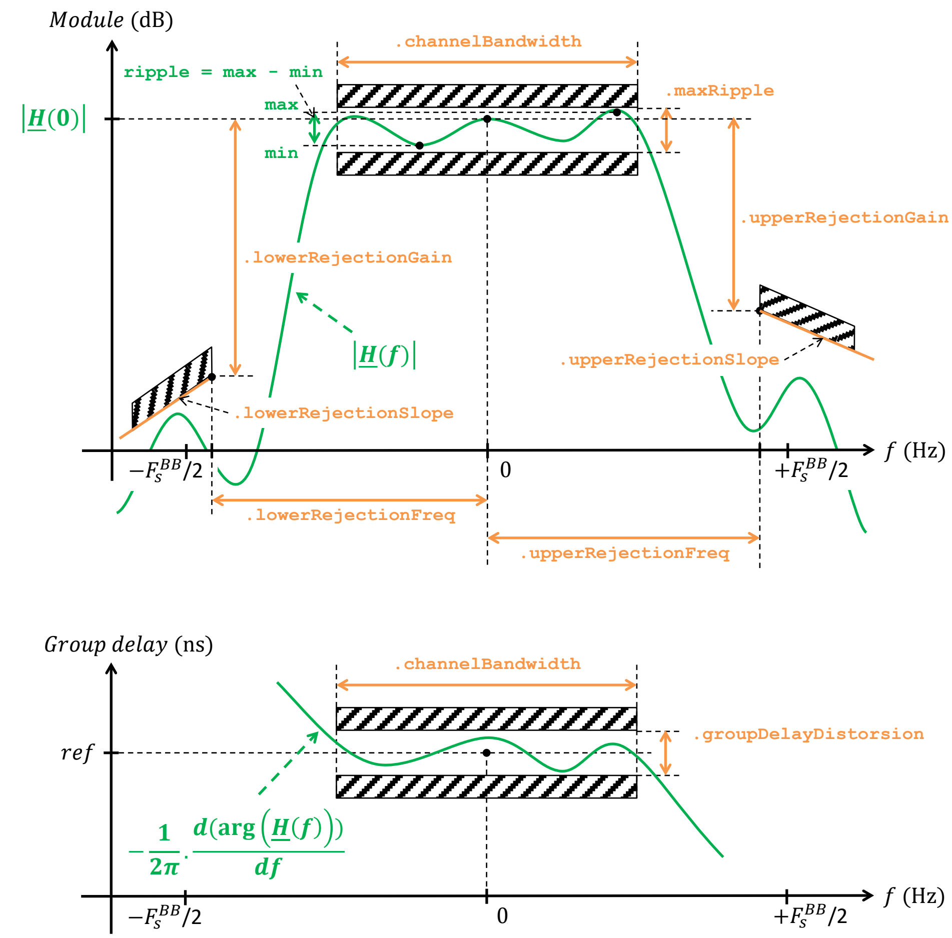

Figure 56 Specification of fields of channel masks

The declaration for **CHANNEL_MASK** **is specified as**, taking the previous figure as reference for specification of the structure's fields:

```
typedef struct {
    // Sampling frequency
    unsigned long basebandSamplingFreq, // in Hz

    // Useful signal
    unsigned long channelBandwidth, // in Hz
    unsigned short ripple, // in thenth of dB
    unsigned short groupDelayDistorsion, // in ns

    // Proximity protection
    unsigned short lowerRejectionFreq, // in Hz
    unsigned short lowerRejectionGain, // in dB
    unsigned short lowerRejectionSlope, // in dB/kHz

    unsigned short upperRejectionFreq, // in Hz
    unsigned short upperRejectionGain, // in dB
    unsigned short upperRejectionSlope // in dB/kHz
} ChannelMask;
```

4.12 Temporal accuracy

A *temporal accuracy property* **is defined as** a *property* that specifies the temporal accuracy of a *transceiver instance*.

The type of a *temporal accuracy property* **is specified as** ***unsigned long***.

Temporal accuracy properties **are specified by** the following table:

Base name	Description	See §
START_TIME_ACC	Applies to: *channels* with at least one *active instance* of a *timely creation service*. Specifies: maximum absolute difference, in nanoseconds (ns), between: ▪ Actual *start time* of a created burst, ▪ *Start time* specified by the *creation operation*.	2.3.2
CURRENT_TIME_ACC	Applies to: *channels* with an *active instance* of **TimeAccess**. Specifies: maximum absolute difference, in nanoseconds (ns), between: ▪ Actual return time of *getCurrentTime()*, ▪ Returned *currentTime* value.	3.1.15
LAST_START_TIME_ACC	Applies to: *channels* with an *active instance* of **TimeAccess**. Specifies: maximum absolute difference, in nanoseconds (ns), between: ▪ Actual *start time* of the last burst, ▪ Returned *lastStartTime* value.	3.1.15

Table 37 Temporal accuracy properties

4.13 Invocation lead time

The *invocation lead time* of a *provide service* primitive **is defined as** the time elapsing, in nanoseconds (ns), between invocation of the primitive by the *radio application* and occurrence within the *transceiver instance* of the future *related event*.

The *invocation lead time property* of a *provide service* **is defined as** a *property* that specifies the minimum value of *invocation lead time* supported by the service.

The type of an *invocation lead time property* **is specified as** `unsigned long`.

Invocation lead time properties **are specified by** the following table:

Base name	Provide service primitive	(Future) Related event	See §
`RELATIVE_MILT`	`RelativeCreation.` *scheduleRelativeBurst()*	*Start time* of the burst.	3.1.4
`ABSOLUTE_MILT`	`AbsoluteCreation.` *scheduleAbsoluteBurst()*	*Start time* of the burst.	3.1.5
`STROBED_MILT`	`StrobedCreation.` *scheduleStrobedBurst()*	*Start time* of the burst.	3.1.6
`TX_PACKET_MILT`	`SamplesTransmission.` *pushTxPacket()*	First sample of the pushed packet is used by *up-conversion*.	3.1.9
`BLOCK_LENGTH_MILT`	`Termination.` *setBlockLength*	*Stop time* of the ongoing *processing phase*. If value of *requestedLength* is not equal to `UndefinedBlockLength`	3.1.7
`TUNING_MILT`	`InitialTuning.` *setTuning()*	Usage of the *creation operation* of the burst by `CreationControl`.	3.1.11
`RETUNING_MILT`	`Retuning.`*retune()*	Start of the `RETUNING` state. If value of *requestedDelay* is not equal to `UndefinedDelay`	3.1.12

Table 38 Invocation lead time properties

4.14 Invocation delay

The *invocation delay* of a *use service* primitive **is defined as** the time elapsing, in nanoseconds (ns), between occurrence within a *transceiver instance* of the past *related event* and invocation of the primitive by the *transceiver instance*.

The *invocation delay property* of a *use service* **is defined as** a *property* that specifies the maximum value of *invocation delay* guaranteed by the service.

The type of an *invocation delay property* **is specified as** `unsigned long`.

Invocation delay properties **are specified by** the following table:

Base name	Use service primitive	(Past) Related event	See §
`PUSH_RX_PACKET_MID`	`SamplesReception.` *pushRxPacket()*	Down-conversion outputs the last sample of the *pushed packet*.	3.1.8
`NOTIFY_EVENT_MID`	`Events.` *notifyEvent()*	The notified error occurs.	3.1.13
`NOTIFY_ERROR_MID`	`Errors.` *notifyError()*	The notified error is detected.	3.1.14
`INDICATE_GAIN_MID`	`GainChanges.` *indicateGain()*	The indicated *Gain* starts to be applied in application of an AGC algorithm decision.	3.1.15

Table 39 Invocation delay properties

4.15 Worst-case execution time (WCET)

The *worst case execution time* (WCET) of a *service* primitive **is defined as** the maximum length of time, in nanoseconds (ns), possibly taken between the invocation and the return of the primitive.

The *WCET property* of a primitive of a *provide service* **is defined as** a *property* that specifies the maximum value of the WCET of the primitive.

The *WCET property* of a primitive of a *use service* **is defined as** a *property* that specifies the maximum value of the WCET of the primitive for correct real-time behavior of the *transceiver instance*.

The type of a *WCET property* **is specified as** ***unsigned long***.

WCET properties of primitives of *provide services* **are specified by** the following table:

Base name	Related primitive	See §
RESET_WCET	Reset::*reset()*	3.1.1
START_SILENCE_WCET	RadioSilence::*startRadioSilence()*	3.1.2
STOP_SILENCE_WCET	RadioSilence::*stopRadioSilence()*	3.1.2
DIRECT_WCET	DirectCreation::*startBurst()*	3.1.3
RELATIVE_WCET	RelativeCreation::*scheduleRelativeBurst()*	3.1.4
ABSOLUTE_WCET	AbsoluteCreation::*scheduleAbsoluteBurst()*	3.1.5
STROBED_WCET	StrobedCreation::*scheduleStrobedBurst()*	3.1.6
BLOCK_LENGTH_WCET	Termination::*setBlockLength()*	3.1.7
STOP_BURST_WCET	Termination::*stopBurst()*	3.1.7
TX_PACKET_WCET	SamplesTransmission::*pushTxPacket()*	3.1.9
RX_PACKETS_LENGTH_WCET	RxPacketsLengthControl::*setRxPacketsLength()*	3.1.10
TUNING_WCET	InitialTuning::*setTuning()*	3.1.11
RETUNING_WCET	Retuning::*retune()*	3.1.12
LOCK_GAIN_WCET	GainLocking::*lockGain()*	3.1.15
UNLOCK_GAIN_WCET	GainLocking::*unlockGain()*	3.1.16
CURRENT_TIME_WCET	TimeAccess::*getCurrentTime()*	3.1.17
LAST_START_TIME_WCET	TimeAccess::*getLastStartTime()*	3.1.17
TRIGGER_STROBE_WCET	ApplicationStrobe::*triggerStrobe()*	3.1.18

Table 40 WCET properties of provide operations

WCET properties of primitives of *use services* **are specified by** the following table:

Base name	Related primitive	See §
RX_PACKET_WCET	SamplesReception::*pushRxPacket()*	3.1.8
EVENTS_WCET	Events::*notifyEvent()*	3.1.13
ERRORS_WCET	Errors::*notifyError()*	3.1.14
GAIN_CHANGE_WCET	GainChanges::*indicateGain()*	3.1.15

Table 41 WCET properties of use operations

5 References

5.1 Referenced documents

[Ref1] *The Fast Guide to Model Driven Architecture*, Cephas Consulting Corp, 2006
http://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf

[Ref2] *Communication Systems*, Simon Haykin, John Whiley & Sons, Inc, 2001

[Ref3] *Digital and Analog Communication Systems*, L.W. Couch, 8th edition, Pearson, 2013

[Ref4] *OMG Unified Modeling Language (OMG UML)*, The Object Management Group, formal/2015-03-01, Version 2.5, March 2015
http://www.omg.org/spec/UML/2.5

[Ref5] *IDL Profiles for Platform-Independent Modeling of SDR Applications*, The Wireless Innovation Forum, WINNF-14-S-0016, Version 2.0.2, 12 June 2015
https://sds.wirelessinnovation.org/specifications-and-recommendations
https://winnf.memberclicks.net/assets/work_products/Specifications/winnf-14-s-0016-v2.0.2.pdf

[Ref6] *Application Interface Definition Language Platform Independent Model Profiles, SCA 4.1 Appendix E-1*, Joint Tactical Networking Center, 20 August 2015
http://www.public.navy.mil/jtnc/sca/Documents/SCAv4_1_Final/SCA_4.1_App_E-1_ApplicationIdlPimProfiles.pdf

[Ref7] *Joint Tactical Radio System Standard Timing Service Application Program Interface*, Joint Tactical Networking Center, Version 1.4.4, 26 June 2013
http://www.public.navy.mil/jtnc/sca/Documents/SCA_APIs/API_1.4.4_20130626_TimingService.pdf

The URLs above were successfully accessed at release date.

END OF THE DOCUMENT