FlexRay Communications System

Protocol Specification

Version 3.0.1

Disclaimer

This specification and the material contained in it, as released by the FlexRay Consortium, is for the purpose of information only. The FlexRay Consortium and the companies that have contributed to it shall not be liable for any use of the specification.

The material contained in this specification is protected by copyright and other types of Intellectual Property Rights. The commercial exploitation of the material contained in this specification requires a license to such Intellectual Property Rights.

This specification may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only.

For any other purpose, no part of the specification may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

Important Information

1. The FlexRay™ specifications V2.1 and V3.0.1 and the corresponding FlexRay™ Conformance Test specifications (hereinafter together "FlexRay™ specifications") have been developed for automotive applications only. They have neither been developed nor tested for non-automotive applications.
2. The FlexRay™ specifications are retrievable on the FlexRay Website www.flexray.com for information purposes only and without obligation.
3. The technical expertise provided in the FlexRay™ specifications is subject to continuous further development. The FlexRay™ specifications serve exclusively as an information source to enable to manufacture and test products which comply with the FlexRay™ specifications ("FlexRay™ compliant products"). Observation of the FlexRay™ specifications does neither guarantee the operability and safety of the FlexRay™ compliant products, nor does it guarantee the safe cooperation of multiple FlexRay™ compliant products with each other or with other products. Therefore, the members of the former FlexRay™ Consortium are not able to assume liability for the operability and safety of such products and the safe cooperation of multiple FlexRay™ compliant products with each other or with other products.
4. The FlexRay™ specifications V3.0.1 were submitted to ISO in order to be published as a standard for road vehicles.

The Core Partners of the FlexRay Consortium are Adam Opel GmbH, Bayerische Motoren Werke AG, Daimler AG, Freescale Halbleiter Deutschland GmbH, NXP B.V., Robert Bosch GmbH, and Volkswagen AG.

Table of Contents

Chapter 9
Controller Host Interface 232

Chapter 1 Introduction

1.1 Scope

The FlexRay communication protocol described in this document is specified for a dependable automotive network. Some of the basic characteristics of the FlexRay protocol are synchronous and asynchronous frame transfer, guaranteed frame latency and jitter during synchronous transfer, prioritization of frames during asynchronous transfer, single or multi-master clock synchronization[1], time synchronization across multiple networks, error detection and signaling, and scalable fault tolerance[2].

1.2 References

1.2.1 FlexRay consortium documents

[EPL10] FlexRay Communications System - Electrical Physical Layer Specification, v3.0.1, FlexRay Consortium, October 2010.

[EPLAN10] FlexRay Communications System - Electrical Physical Layer Application Notes, v3.0.1, FlexRay Consortium, October 2010.

[PCT10] FlexRay Communications System - Protocol Conformance Test Specification, v3.0.1, FlexRay Consortium, October 2010.

1.2.2 Non-consortium documents

[Cas93] G. Castagnoli, S. Bräuer, and M. Herrmann, "Optimization of Cyclic Redundancy-Check Codes with 24 and 32 Parity Bits", IEEE Transactions on Communications, vol. 41, pp. 883-892, June 1993.

[Koo02] P. Koopman, "32-bit Cyclic Redundancy Codes for Internet Applications", Proceedings of the International Conference on Dependable Systems and Networks (DSN 2002), Washington DC, pp. 459-468. June 2002.

[Ung09] J. Ungermann, “On Clock Precision Of FlexRay Communication Systems”, Dec. 2009, available at: http://www.flexray.com

[Wad01] T. Wadayama, “Average Distortion of Some Cyclic Codes”, web site available at: http://www-tkm.ics.nitech.ac.jp/~wadayama/distortion.html

[Wel88] J. L. Welch and N. A. Lynch, "A New Fault-Tolerant Algorithm for Clock Synchronization", Information and Computation, vol. 77, no. 1, pp. 1-36, April 1988.

[Z100] ITU-T Recommendation Z.100 (03/93), Programming Languages - CCITT Specification and Description Language (SDL), International Telecommunication Union, Geneva, 1993.

[1] Multi-master clock synchronization refers to a synchronization that is based on the clocks of several (two or more) synchronization masters or sync nodes.

[2] Scalable fault tolerance refers to the ability of the FlexRay protocol to operate in configurations that provide various degrees of fault tolerance (for example, single or dual channel clusters, clusters with many or few sync nodes, etc.).

1.3 Revision history

Vers.	Date	Changes
2.0	30-Jun-2004	First public release.
2.1	May 2005	The SDL processes in Chapter 3 (Coding) have been restructured. Appendix B has been almost completely rewritten. BG references and BGSM chapter (former chapter 10) have been removed. Specific significant changes to CHI: • channel dependency of *pMicroInitialOffset[Ch]* and *pMacroInitialOffset[Ch]* has been introduced in 9.3.1.1.2 • *pDecodingCorrection* has been added in CHI in 9.3.1.1.2 • error indicator in CHI has been added in 9.3.1.3.3 • *vSyncFramesEven*/*Odd*/*A*/*B* have been removed in 9.3.1.3.4 • indicators for *zLastDynTxSlot* have been added in 9.3.1.3.9 The status variable *vPOC!StartupState* has been introduced in the CHI in 9.3.1.3.1. The variable is reset in Figure 2-6 to Figure 2-8 and set in Figure 7-11 to Figure 7-19. Figure 6-10 was split into two figures (now Figure 6-10 and Figure 6-11). The following figures have modifications that changed the operation of the protocol: Figure 2-8, Figure 5-21, Figure 6-8, Figure 6-16 (former 6-15), Figure 6-17 (former 6-16), Figure 7-3, Figure 7-11, Figure 7-19, Figure 8-4, Figure 8-8, Figure 8-10, Figure 8-11, Figure 8-15, and Figure 8-17. The text related to these figures has also been updated. Numerous non-technical corrections and clarifications were made throughout the document.
2.1 Rev A	Dec 2005	Use of SDL priority input to resolve certain race conditions (see section 1.7.3.4 and Figure 5-21 and Figure 8-8) Re-arrangement of SDL to eliminate the use of SDL "enabling condition" structure (Figure 6-10, Figure 6-11, Figure 7-3, Figure 7-4, and Figure 7-5) Update of *zLastDynTxSlot* (Figure 5-13, Figure 5-20, Figure 5-21, and Figure 5-22) Re-arrangement of channel idle detection (Figure 2-9, Figure 3-15, Figure 3-16, Figure 3-17, Figure 3-18, Figure 3-25, Figure 3-36, Figure 3-37, Figure 7-3, and Figure 7-11) Introduction of new "a" class of variables (see section 1.6.1) Replacement of the bit counter by a timer in Figure 3-23 Explicit export of all CHI variables Extension of the color coding to SDL signals (see section 1.6.2) Rework of Appendix B Numerous non-technical corrections and clarifications were made throughout the document.

Table 1-1: Revision history

Vers.	Date	Changes
3.0	Dec 2009	Ongoing transmission at end of dynamic segment no longer leads to a fatal protocol error. A null frame received with the *PPIndicator* set is marked with a Content Error. Frames are discarded if first FES bit is incorrectly received. Defined behavior of TxD pin in case of aborted transmissions. Support of 2.5 and 5 Mbit/s. Wakeup pattern detection aligned to reception defined in EPL specification. *pKeySlotID* can be configured to zero for a node that does not have a key slot. TxEN and TxD are not switched off simultaneously. Reception is ignored for a configurable time after transmission. Within the symbol window a new symbol, the Wakeup During Operation Pattern (WUDOP), can be transmitted to support wakeup during normal operation. Receivers now accept a TSS 2 bits longer than the nominal duration. The idle detection mechanism is based on time duration instead of bit duration. The READY command was renamed to IMMEDIATE_READY. The HALT command was renamed to DEFERRED_HALT command. Two new commands were added: DEFERRED_READY and CLEAR_DEFERRED. Two new synchronization methods, TT-L and TT-E, were added to the existing TT-D synchronization method. Slot multiplexing in the static segment is allowed for all slots except for key slots. The cycle counter wraparound value was made configurable. The cycle counter filtering was extended to include repetition values of 5, 10, 20, 40, and 50. Rework of Appendix B (including but not limited to): • precision formulas • microtick distribution error • removal of 100 ns microtick • internal controller delays are now part of the propagation delay • ranges for external clock correction parameters were increased • maximum transmission duration aligned with EPL specification • optimized constraint for static slot size • a specific clock deviation, *gClockDevationMax*, for a given cluster was added, replacing *cClockDeviationMax* in the configuration constraints • *gClockDevationMax* is now used as a frequency deviation instead a clock period deviation • removal of 1:4 relation between sample tick and microtick, i.e., support of 50 ns microtick @ 10 Mbit/s is no longer required. All remaining sample tick-microtick combinations are now mandatory. • The parameter *gOffsetCorrectionStart* was renamed to *pOffsetCorrectionStart*. • *pdMaxDrift* was replaced by *pRateCorrectionOut*.

Table 1-1: Revision history

Vers.	Date	Changes
		• Various updates to remain consistent with the names and ranges of parameters in the EPL specification and EPLAN document. • Modifications to a number of configuration constraints to take into account the possibility of ringing. • Configuration constraints for TT-L and TT-E clusters were added. Numerous modifications to the MAC process to improve robustness against noise that occurs in the dynamic segment. New CHI parameters, *vDynResyncAttempt[A]* and *vDynResyncAttempt[B]* were introduced. The default configuration now prevents transmission and reception. Updates to the behavior of wakeup and startup of single channel devices when the noise timer expires. Frames received in transmission slots are marked as invalid. Support for the NM vector became mandatory. It is configurable if the NM Vector is exported to the CHI at the end of the static segment or at the end of the cycle. Number of valid startup frame pairs is exported to the CHI. *PPIndicator* was removed from the CHI slot status information (but remains in the frame contents data). A frame transmitted indicator was added to the CHI transmit buffer status. A corresponding element (*vSS!FrameSent*) was added to the protocol engine's slot status. The optional relative timer was removed. A second absolute timer became mandatory. Additional interrupts became mandatory. The required behaviour of the interrupt service was specified in more detail. At least one FIFO buffer is now mandatory and the FIFO filter criteria were specified. The slot status flag *vSS!TxConflict* was added to the aggregated channel status data. Clarification that a received buffer configured for both channels must also store the source of the frame contents-related data. Clarification in which POC states the configuration of message buffers and the transmission slot assignment list is either allowed or forbidden. Configuring or reconfiguring a buffer clears the payload data valid and slot status updated flags. Transition out of *POC:ready* into one of the states associated with startup clears the payload data valid and slot status updated flags on receive buffers. Default initialization of indicators. Power level and time thresholds were removed from Chapter 2. The wakeup application notes were moved from Chapter 7 to the new Appendix C. Numerous smaller technical corrections, non-technical corrections and clarifications were made throughout the document.

Table 1-1: Revision history

Vers.	Date	Changes
3.0.1	Oct 2010	Initialization of the time measurement values for the clock synchronization in the TT-E mode. See Fig. 8-5 and Fig. 8-7. Description of the basic FIFO behavior in more detail (see 9.3.2.10.1). Frame export to CHI extented to nullframes (Fig. 6-9). Description of the concepts of buffer enabling and buffer locking in more detail. See 9.3.2.5.4. Reset of the control variables *vTransmitMTS_A*, *vTransmitWUDOP_A*, *vTransmitMTS_B*, *vTransmitWUDOP_B*, *vExternOffsetControl*, and *vExternRate-Control* during or after a transition into *POC:ready* or *POC:halt*. A fix that a time gateway sink node is able to make the transition from the state *POC:normal passive* to *POC:normal active* if the time gateway source node makes the transition from *POC:normal passive* to *POC:normal active*. See Fig. 8-22. Section A.1.1 was deleted. Fig. 3-8 was modified. The requirements for the export of *vStartupPairs* were modified to address the TT-E mode. See 9.3.1.3.5. Distinction between internal and external signals for TxD, TxEN, and RxD was added. See 1.12.4 and Fig. 1-13. Exceptions for TT-E coldstart nodes added for the data and slot status handling in the first slot in the first cycle after a TT-E coldstart node's transition from the *POC:external startup* state to the *POC:normal active* state (see Figures 5-13 and 5-17 and sections 9.3.2.8.2.2, 9.3.2.9, and 9.3.2.10.1.1).

Table 1-1: Revision history

1.4 Terms and definitions

application data

data produced and/or used by application tasks. In the automotive context the term 'signal' is often used for application data exchanged among tasks.

bus

a communication system topology in which nodes are directly connected to a single, common communication media (as opposed to connection through stars, gateways, etc.). The term bus is also used to refer to the media itself.

bus driver

an electronic component consisting of a transmitter and a receiver that connects a communication controller to one communication channel.

channel

see communication channel.

channel idle

the condition on the physical transmission medium when no node is transmitting, as perceived by each individual node in the network. Note that detection of channel idle occurs some time after all nodes have actually stopped transmitting (due to idle detection times, channel effects, ringing, etc.).

clique

set of communication controllers having the same view of certain systems properties, e.g., the global time value or the activity state of communication controllers.

cluster

a communication system of multiple nodes connected via at least one communication channel directly (bus topology), by active stars (star topology) or by a combination of bus and star connections (hybrid topologies).

coldstart node

a node capable of initiating the communication startup procedure on the cluster by sending startup frames. TT-D coldstart nodes, TT-L coldstart nodes, and TT-E coldstart nodes are all considered to be coldstart nodes. By definition, all coldstart nodes are also sync nodes.

communication channel

the inter-node connection through which signals are conveyed for the purpose of communication. The communication channel abstracts both the network topology (bus, star or hybrid), as well as the physical transmission medium.

communication controller (CC)

an electronic component in a node that is responsible for implementing the protocol aspects of the FlexRay communications system.

communication cycle

one complete instance of the communication structure that is periodically repeated to comprise the media access method of the FlexRay system. The communication cycle consists of a static segment, an optional dynamic segment, an optional symbol window, and a network idle time.

communication slot

an interval of time during which access to a communication channel is granted exclusively to a specific node for the transmission of a frame with a frame ID corresponding to the slot. FlexRay distinguishes between static communication slots and dynamic communication slots.

cycle-dependent slot assignment

method of assigning, for a given channel, an individual slot (identified by a specific slot number and a specific cycle counter number) or a set of slots (identified by a specific slot number and a set of communication cycle numbers) to a node.

cycle-independent slot assignment

method of assigning, for a given channel, the set of all communication slots having a specific slot number to a node (i.e., on the given channel, slots with the specific slot number are assigned to the node in all communication cycles).

cycle number

A positive integer used to identify a communication cycle. The cycle number of each communication cycle is one greater than the cycle number of the previous cycle, except in cases where the previous cycle had the maximum cycle number value, in which case the cycle number has the value of zero. The cycle number of the first cycle is, by definition, zero.

cycle time

the time within the current communication cycle, expressed in units of macroticks. Cycle time is reset to zero at the beginning of each communication cycle.

dynamic segment

portion of the communication cycle where the media access is controlled via a mini-slotting scheme, also known as Flexible Time Division Multiple Access (FTDMA). During this segment access to the media is dynamically granted on a priority basis to nodes with data to transmit.

dynamic slot / dynamic communication slot

an interval of time within the dynamic segment of the communication cycle consisting of one or more minislots during which access to a communication channel is granted exclusively to a specific node for transmission of a frame with a frame ID corresponding to the slot. In contrast to a static communication slot, the duration of a dynamic communication slot may vary depending on the length of the frame. If no frame is sent, the duration of a dynamic communication slot equals that of one minislot.

frame

a structure used by the communication system to exchange information within the system. A frame consists of a header segment, a payload segment and a trailer segment. The payload segment is used to convey application data.

frame identifier

the frame identifier defines the slot position in the static segment and defines the priority in the dynamic segment. A lower identifier indicates a higher priority.

gateway

a node that is connected to two or more independent communication networks that allows information to flow between the networks.

global time

combination of cycle counter and cycle time.

Hamming distance

the minimum distance (i.e., the number of bits which differ) between any two valid code words in a binary code.

host

the part of an ECU where the application software is executed, separated by the CHI from the FlexRay protocol engine.

implementation dependent

Behavior that, subject to restrictions in the specification, may be chosen by an implementation designer. Implementation dependent behavior must be described in detail in the documentation of an implementation.

key slot

a static slot that is used by a node to transmit sync and startup frames. The key slot is also the slot used to transmit when the node is operating in key slot only mode.

macrotick

an interval of time derived from the cluster-wide clock synchronization algorithm. A macrotick consists of an integral number of microticks. The actual number of microticks in a given macrotick is adjusted by the clock synchronization algorithm. The macrotick represents the smallest granularity unit of the global time.

microtick

an interval of time derived directly from the CC's oscillator (possibly through the use of a prescaler). The microtick is not affected by the clock synchronization mechanisms, and is thus a node-local concept. Different nodes can have microticks of different duration.

minislot

an interval of time within the dynamic segment of the communication cycle that is of constant duration (in terms of macroticks) and that is used by the synchronized FTDMA media access scheme to manage media arbitration.

non-coldstart node

a node that is not capable of initiating the communication startup procedure (i.e., does not transmit startup frames).

non-sync node

a node that is not configured to transmit sync frames.

non-synchronized operation

operation of a node when the node does not have a notion of FlexRay time, i.e., has no knowledge of slot identifier, slot boundaries, cycle counter, or segment boundaries.

network

the combination of the communication channels that connect the nodes of a cluster.

network topology

the arrangement of the connections between the nodes. FlexRay supports bus, star, cascaded star, and hybrid network topologies.

node

a logical entity connected to the network that is capable of sending and/or receiving frames.

null frame

a frame that contains no usable data in the payload segment. A null frame is indicated by a bit in the header segment, and all data bytes in the payload segment are set to zero.

physical communication link

an inter-node connection through which signals are conveyed for the purpose of communication. All nodes connected to a given physical communication link share the same signals (i.e., they are not connected through repeaters, stars, gateways, etc.). Examples of a physical communication link include a bus network or a point-to-point connection between a node and a star. A communication channel may be constructed by combining one or more physical communication links together using stars.

precision

the worst-case deviation between the corresponding macroticks of any two synchronized nodes in the cluster.

slot

see communication slot

slot ID (identifier)

see slot number

slot multiplexing

the technique of assigning, for a given channel, slots having the same slot identifier to different nodes in different communication cycles.

slot number

number used to identify a specific slot within a communication cycle.

star

a device that allows information to be transferred from one physical communication link to one or more other physical communication links. A star duplicates information present on one of its links to the other links connected to the star. A star can be either passive or active. For the purposes of this specification, all usages of the term "star" are references to an active star as described in [EPL10].

startup frame

FlexRay frame whose header segment contains an indicator that integrating nodes may use time-related information from this frame for initialization during the startup process. Startup frames are always also sync frames.

static slot / static communication slot

an interval of time within the static segment of the communication cycle that is constant in terms of macroticks and during which access to a communication channel is granted exclusively to a specific node for transmission of a frame with a frame ID corresponding to the slot. Unlike a dynamic communication slot, each static communication slot contains a constant number of macroticks regardless of whether or not a frame is sent in the slot.

static segment

portion of the communication cycle where the media access is controlled via a static Time Division Multiple Access (TDMA) scheme. During this segment access to the media is determined solely by the progression of time.

sync frame

FlexRay frame whose header segment contains an indicator that the deviation measured between the frame's arrival time and its expected arrival time should be used by the clock synchronization algorithm.

sync node

a node configured to transmit sync frames. Coldstart nodes and TT-D non-coldstart sync nodes are considered to be sync nodes.

synchronized operation

operation of a node when the node has a notion of FlexRay time, i.e., has knowledge of slot identifier, slot boundaries, cycle counter, and segment boundaries.

time gateway

a pair of nodes attached to different clusters connected by a time gateway interface.

time gateway interface

an interface used by a time gateway source node to provide timing information for a time gateway sink node.

time gateway sink node

a node configured as TT-E coldstart node, which is connected via a time gateway interface to a time gateway source node. The time gateway sink node receives timing information from the time gateway source node.

time gateway source node

a node connected via a time gateway interface to a time gateway sink node. The time gateway source node provides timing information for the time gateway sink node.

time sink cluster

a cluster using the TT-E synchronization method. The term emphasizes that the TT-E coldstart nodes of this cluster receive their timing from another cluster.

time source cluster

a cluster that provides the timing information for a time sink cluster.

transmission slot assignment list

structure identifying the set of all slots assigned to a node for transmission.

TT-D cluster

a cluster in which the clock synchronization uses the TT-D synchronization method. A TT-D cluster consists of two or more TT-D coldstart nodes, zero or more TT-D non-coldstart sync nodes, and zero or more non-sync nodes.

TT-D coldstart node

a coldstart node operating in a TT-D cluster. This node has only a single key slot and sends a startup/sync frame in the configured key slot in each cycle on each configured channel.

TT-D non-coldstart sync node

a node that is configured to transmit sync frames but is not capable of initiating the communication startup procedure (i.e., does not send startup frames).

TT-D synchronization method

a method of clock synchronization in which the clock synchronization is derived in a distributed manner from two or more sync nodes. Two or more coldstart nodes are required to start up a cluster using this synchronization method.

TT-E cluster

a cluster in which the clock synchronization uses the TT-E synchronization method. A TT-E cluster consists of one or more TT-E coldstart nodes and zero or more non-sync nodes.

TT-E coldstart node

a coldstart node operating in a TT-E cluster. This node has two key slots and sends startup/sync frames in both configured key slots in each cycle on each configured channel. A TT-E coldstart node is a time gateway sink (i.e., is configured for external synchronization) and bases its timebase on the clock sync information derived from the time source cluster as delivered by the time gateway interface.

TT-E synchronization method

a method of clock synchronization in which the clock synchronization is derived directly from the clock synchronization of another FlexRay cluster. In this method a single coldstart node is capable of starting up the cluster.

TT-L cluster

a cluster in which the clock synchronization uses the TT-L synchronization method. A TT-L cluster consists of one TT-L coldstart node and one or more non-sync nodes.

TT-L coldstart node

a coldstart node operating in a TT-L cluster. This node has two key slots and sends startup/sync frames in both configured key slots in each cycle on each configured channel.

TT-L synchronization method

a method of clock synchronization in which the clock synchronization is derived from the local clock of a single sync node, and in which a single coldstart node starts up the cluster.

1.5 Acronyms and abbreviations

µs	Microsecond
µT	Microtick
AP	Action Point
BD	Bus Driver
BIST	Built-In Self Test
BITSTRB	Bit Strobing Process
BSS	Byte Start Sequence
CAS	Collision Avoidance Symbol

CC	Communication Controller
CE	Communication Element
CHI	Controller Host Interface
CHIRP	Channel Idle Recognition Point
CODEC	Coding and Decoding Process
CRC	Cyclic Redundancy Code
CSP	Clock Synchronization Process
CSS	Clock Synchronization Startup Process
DTS	Dynamic Trailing Sequence
ECU	Electronic Control Unit, same as node
EMI	Electromagnetic Interference
ERRN	Error Not signal
FES	Frame End Sequence
FIFO	First In First Out
FSP	Frame and Symbol Processing
FSS	Frame Start Sequence
FTDMA	Flexible Time Division Multiple Access
FTM	Fault-Tolerant Midpoint
ID	Identifier
INH1	Inhibit signal
MAC	Media Access Control Process
MT	Macrotick
MTG	Macrotick Generation Process
MTS	Media Access Test Symbol
NIT	Network Idle Time
NM	Network Management
POC	Protocol Operation Control
RxD	Receive data signal from bus driver
SDL	Specification and Description Language
SPI	Serial Peripheral Interface
ST	Sampletick
STBN	Standby Not signal
SW	Symbol Window
TDMA	Time Division Multiple Access
TRP	Time Reference Point
TSS	Transmission Start Sequence

TT-D	Time-Triggered Local Distributed
TT-E	Time-Triggered External
TT-L	Time-Triggered Local Master
TxD	Transmit Data signal from CC
TxEN	Transmit Data Enable Not signal from CC
WUDOP	Wakeup During Operation Pattern
WUP	Wakeup Pattern
WUPDEC	Wakeup Pattern Decoding Process
WUS	Wakeup Symbol

1.6 Notational conventions

1.6.1 Parameter prefix conventions

<variable> ::= <prefix_1> [<prefix_2>] Name

<prefix_1>::= a | c | v | g | p | z

<prefix_2>::= d | s

Naming Convention	Information Type	Description
a	Auxiliary Parameter	Auxiliary parameter used in the definition or derivation of other parameters or in the derivation of constraints.
c	Protocol Constant	Values used to define characteristics or limits of the protocol. These values are fixed for the protocol and cannot be changed.
v	Node Variable	Values that vary depending on time, events, etc.
g	Cluster Parameter	Parameter that must have the same value in all nodes in a cluster, is initialized in the *POC:default config* state, and can only be changed while in the *POC:config* state.
p	Node Para-meter	Parameter that may have different values in different nodes in the cluster, is initialized in the *POC:default config* state, and can only be changed while in the *POC:config* state.
z	Local SDL Process Variable	Variables used in SDL processes to facilitate accurate representation of the necessary algorithmic behavior. Their scope is local to the process where they are declared and their existence in any particular implementation is not mandated by the protocol.

Table 1-2: Parameter prefix 1.

Naming Convention	Information Type	Description
d	Time Duration	Value (variable, parameter, etc.) describing a time duration, the time between two points in time.
s	Set	Set of values (variables, parameters, etc.).

Table 1-3: Parameter prefix 2.

1.6.2 Color coding

Throughout the text several types of items are highlighted through the use of an italicized color font.

Parameters, constants and variables are highlighted with *blue italics*. An example is the parameter *gdStaticSlot*. This convention is not used within SDL diagrams, as it is assumed that such information is obvious. The meaning of the prefixes of parameters, constants, and variables is described in section 1.6.1.

SDL states are highlighted in *green italics*. An example is the SDL state *POC:normal active*. This highlighting convention is not used within SDL diagrams. Further notational conventions related to SDL states are described in section 1.7.2.

SDL signals are highlighted in *red italics*. An example is the SDL signal *CHIRP on A*. Again, this convention is not used within the SDL diagrams themselves as the fact that an item is an input or output signal should be obvious.

1.6.3 Implementation dependent behavior

While this specification defines the required behavior of a FlexRay implementation in many respects, there are various decisions on the particulars of an implementation that, for flexibility reasons, are left up to the implementation designers. This specification defines the term "implementation dependent" to have the following meaning:

> A behavior (or a parameter or characterization of a behavior, such as a default value) that, subject to restrictions contained in this specification, may be chosen by an implementation designer.
> Implementation dependent behavior may vary from implementation to implementation, but the specific behavior must be described in detail in the documentation of the implementation.

1.7 SDL conventions

1.7.1 General

The FlexRay protocol mechanisms described in this specification are presented using a graphical method loosely based on the Specification and Description Language (SDL) technique described in [Z100]. The intent of this description is not to provide a complete executable SDL model of the protocol mechanisms, but rather to present a reasonably unambiguous description of the mechanisms and their interactions. This description is intended to be read by humans, not by machines, and in many cases the description is optimized for understandability rather than exact syntactic correctness.

The SDL descriptions in this specification are behavioral descriptions, not requirements on a particular method of implementation. In many cases the method of description was chosen for ease of understanding rather than efficiency of implementation. An actual implementation should have the same behavior as the SDL description, but it need not have the same underlying structure or mechanisms.

Several SDL diagrams have textual descriptions intended to assist the reader in understanding the behavior depicted in the SDL diagrams. Some technical details are intentionally omitted from these explanations. Unless specifically mentioned, the behavior depicted in the SDL diagrams takes precedence over any textual description.

In SDL, transitions between states, and any processing that takes place along the paths involving these transitions, is assumed to take place in zero time. The descriptions of the protocol mechanisms rely on this zero time assumption to specify the proper behavior of an implementation. Transitions and processing in a real implementation will not take place in zero time. The implementation designer must comprehend any discrepancy between the implicit zero time assumption in the SDL description and the actual time taken in the chosen implementation technology and ensure that the implementation's behavior is consistent with the behavior described in the SDL.

1.7.2 SDL notational conventions

States that exist within the various SDL processes are shown with the state symbol shaded in light gray. These states are named with all lowercase letters. Acronyms or proper nouns that appear in a state name are capitalized as appropriate. Examples include the states "wait for sync frame" and "wait for CE start".

SDL states that are referenced in the text are prefixed with an identification of the SDL process in which they are located (for example, the state *POC:normal active* refers to the "normal active" state in the POC process). This convention is not used within the SDL diagrams themselves, as the process information should be obvious.

The definitions of an SDL process are often spread over several different figures. The caption of each figure that contains SDL definitions indicates to which SDL process the figure belongs.

1.7.3 SDL extensions

The SDL descriptions in this specification contain some constructs that are not a part of normal SDL. Also, some mechanisms described with constructs that are part of normal SDL expect that these constructs behave somewhat differently than is described in [Z100]. This section documents significant deviations from "standard" SDL.

1.7.3.1 Microtick, macrotick and sample tick timers

The representation of time in the FlexRay protocol is based on a hierarchy that includes microticks and macroticks (see Chapter 8 for details). Several SDL mechanisms need timers that measure a certain number of microticks or macroticks. This specification makes use of an extension of the SDL 'timer' construct to accomplish this.

An SDL definition of the form

```
µT timer
```

defines a timer that counts in terms of microticks. This behavior would be similar to that of an SDL system whose underlying time unit is the microtick.

An SDL definition of the form

```
MT timer
```

defines a timer that counts in terms of macroticks. Note that a macrotick timer uses the corrected macroticks generated by the macrotick generation process. Since the duration of a macrotick can vary, the duration of these timers can also vary, but the timers themselves remain synchronized to the macrotick-level timebase of the protocol.

In all other respects both of these constructs behave in the same manner as normal SDL timers.

In addition to the above, several SDL mechanisms used in the description of encoding make use of a timer that measures a certain number of ticks of the bit sample clock. An SDL definition of the form

```
ST timer
```

defines a timer that counts in terms of ticks of the bit sample clock (i.e., sample ticks). This behavior would be similar to that of an SDL system whose underlying time unit is the sample tick. In all other respects this construct behaves in the same manner as a normal SDL timer.

There is a defined relationship between the "ticks" of the microtick timebase and the sample ticks of bit sampling. Specifically, a microtick consists of an integral number, *pSamplesPerMicrotick*, of sample ticks. As a result, there is a fixed phase relationship between the microtick timebase and the ticks of the sample clock.

The time expression of a timer is defined in [Z100] by:

```
<Time expression> = now + <Duration constant expression>
```

In this specification the time expression is used in the following simplified way:

```
<Time expression> = <Duration constant expression>
```
[3]

1.7.3.2 Microtick behavior of the 'now' - expression

The behavioral descriptions of various aspects of the FlexRay system require the ability to take "time-stamps" at the occurrence of certain events. The granularity of these timestamps is one microtick, and the timestamps taken by different processes need to be taken against the same underlying timebase. This specification makes use of an extension of the SDL concept of time to facilitate these timestamps.

This specification assumes the existence of an underlying microtick timebase. This timebase, which is available to all processes, contains a microtick counter that is started at zero at some arbitrary epoch assumed to occur before system startup. As time progresses, this timebase increments the microtick counter without bound[4]. Explicit use of the SDL 'now' construct returns the value of this microtick counter. The difference between the timestamps of two events represents the number of microticks that have occurred between the events.

1.7.3.3 Channel-specific process replication

The FlexRay protocol described in this specification is a dual channel protocol. Several of the mechanisms in the protocol are replicated on a channel basis, i.e., essentially identical mechanisms are executed, one for channel A and one for channel B. This specification only provides SDL descriptions for the channel-specific processes on channel A - it is assumed that whenever a channel-specific process is defined for channel A there is another, essentially identical, process defined for channel B, even though this process is not explicitly described in the specification.

Channel-specific processes have names that include the identity of their channel (for example, "Clock synchronization startup process on channel A [CSS_A]"). In addition, some signals that leave a channel-specific process have signal names that include the identity of their channel (for example, the signal *integration aborted on A*).[5]

1.7.3.4 Handling of priority input symbols

The SDL language contains certain ambiguities regarding the order of execution of processes if multiple processes have input queues that are not empty. For example, the usage of timers and clock oscillator inputs causes multiple processes to be eligible for execution at the beginning of clock edges. Generally, this poses no problem for the FlexRay specification, but for certain special cases it is not possible to specify the required behavior in an unambiguous way without additional language constructs.

[3] If the duration time expression is zero or negative then the timer is started and expires immediately.
[4] This is in contrast to the *vMicrotick* variable, which is reset to zero at the beginning of each cycle.
[5] It is also possible for a signal leaving a channel-specific process to have a name that does not identify the channel. In such cases, a process that receives the SDL signal should behave identically regardless of which process sent the signal (i.e., the process receiving the signal effectively OR's the signals from all of the sending processes).

To resolve these situations the SDL priority input symbol is used, but with a slightly extended meaning. Whenever an input priority symbol is used, no other exit path of this state may be taken unless it is impossible that the priority input could be triggered on the current microtick clock edge. Effectively, the execution of the process in question is stalled until all other processes have executed. Should multiple processes be in a state where they are sensitive to a priority input, all are executed last and in random order. The message queue is handled in the standard way, i.e. the signal triggering the priority input is removed from the queue while any signals placed before or after are preserved for the succeeding state.

1.7.3.5 Signals to non-instantiated processes

In various portions of the SDL behavioral descriptions the SDL sends signals that are received by a process that in some conditions may not be instantiated. As an example, the SDL in Figure 2-12 generates the signal *CODEC control on B (NORMAL)* even if the node is only attached to channel A (implying that the CODEC_B process is not instantiated). The sending of these signals should not be interpreted as requiring that processes that receive the signal should be instantiated - in such cases these signals should simply be ignored. This convention holds even if the only process that consumes the signal in question is a process that is not already instantiated.

1.7.3.6 Exported and imported signals

Certain features of the FlexRay protocol require that certain direct communication between the two communication controllers of a time gateway is modeled within the SDL diagrams (for example, see section 1.10.3). Signals marked with the EXP keyword are distributed within the local communication controller like any other SDL signal, but are in addition are also forwarded to a second communication controller that is represented by a separate instance of the SDL diagrams. On the receiving end, these exported signals can be received by input symbols marked with the IMP keyword. Input symbols marked in this way are only sensitive to signals emitted with the EXP keyword of the *other* communication controller.

1.8 Bit rates

The FlexRay Communications System specifies three standard bit rates - 10 Mbit/s (corresponding to a nominal bit duration, *gdBit*, of 100 ns), 5 Mbit/s (corresponding to a *gdBit* of 200 ns), and 2.5 Mbit/s (corresponding to a *gdBit* of 400 ns). In order to be considered FlexRay conformant, a protocol implementation is required to support all three standard bit rates.

1.9 Roles of a node in a FlexRay cluster

There are three distinct roles a node can perform.

1. The role of a sync node enables a node to actively participate in the clock synchronization algorithm performed by the cluster. Sync nodes transmit sync frames that are evaluated by all nodes of the cluster to perform an alignment of clock rate, that effectively determines the cycle length, and clock offset, that effectively determines the position of the cycle start.
2. The role of a coldstart node enables a node to initiate the communication. Coldstart nodes are allowed to start transmitting startup frames in the non-synchronized state with the intent of establishing a schedule. Nodes integrate onto that new schedule by evaluating the content and timing of the received startup frames. A coldstart node is always also a sync node and a startup frame is always also a sync frame.
3. A node that is neither a coldstart node nor a sync node is referred to as non-sync node. It performs no special task.

1.10 Synchronization methods

A FlexRay node supports three different synchronization modes. The behavior of the cluster depends on the employed synchronization mode of the nodes in the cluster.

1.10.1 TT-D synchronization method

A cluster in which the coldstart nodes use the TT-D synchronization method is a TT-D cluster. The TT-D synchronization method uses a distributed algorithm to reduce the effect of any single failure. No critical task depends on any single node. A distributed startup instigated and carried through by two to fifteen coldstart nodes mitigates many adverse effects a single faulty coldstart node can have (see Chapter 7). A distributed clock synchronization algorithm actively driven by two to fifteen sync nodes is robust against a number of Byzantine faults depending on the number of currently active sync nodes (see Chapter 8).

The advantage of the TT-D synchronization method over the others is increased fault-tolerance.

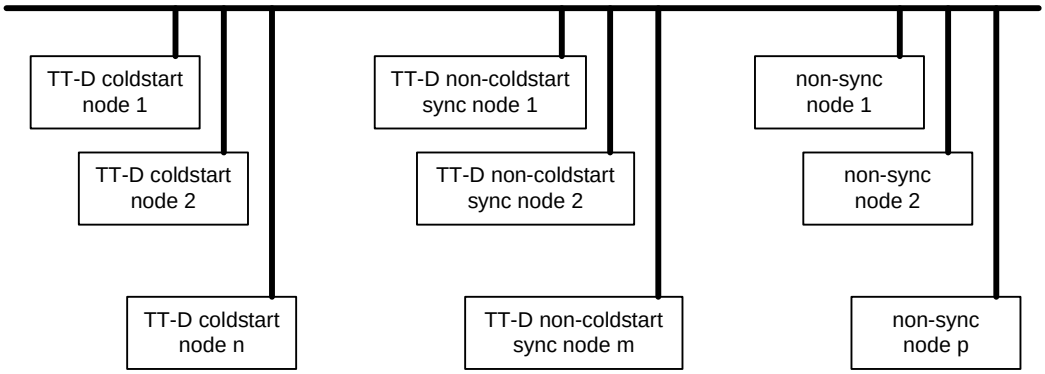

Figure 1-1: TT-D cluster.

Figure 1-1 shows the configuration of nodes in a TT-D cluster. The number of TT-D coldstart nodes n must be equal to or greater than 2 and the sum of the number of TT-D coldstart nodes n and the number of TT-D non-coldstart sync nodes m must be equal to or less than 15. The number of non-sync nodes p is not limited by the protocol.

1.10.2 TT-L synchronization method

A cluster in which the sole coldstart node uses the TT-L synchronization method is a TT-L cluster. The TT-L synchronization method is a modification of the TT-D synchronization method that reduces the number of required coldstart nodes from two to one. The single TT-L coldstart node in a TT-L cluster essentially behaves like two regular TT-D coldstart nodes by transmitting two startup frames. In this way non-sync nodes of the TT-L cluster will behave as if they were placed in a TT-D cluster with two TT-D coldstart nodes regularly transmitting their startup/sync frames and will integrate and operate normally, unaware of the fact that the two frames they receive actually come from the same node. The schedule and timing of such a TT-L cluster will depend entirely on the single TT-L coldstart node.

The advantages of the TT-L synchronization method are a reduced system complexity, a slightly reduced startup time, and an improved precision.

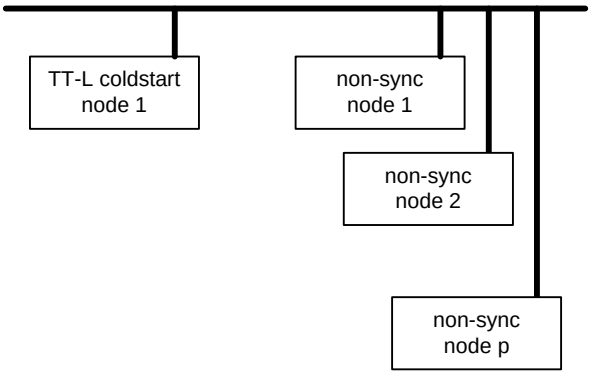

Figure 1-2: TT-L cluster.

Figure 1-2 shows the configuration of nodes in a TT-L cluster. There exists exactly one TT-L coldstart node. The number of non-sync nodes p is not limited by the protocol.

1.10.3 TT-E synchronization method

A cluster in which the coldstart nodes use the TT-E synchronization method is a TT-E cluster. The primary intent of the TT-E synchronization method is to synchronize the schedule of the TT-E cluster, also called a time sink cluster, to a second FlexRay cluster, which is referred to as time source cluster. To that end, each TT-E coldstart node, also called a time gateway sink node, must be paired with a node of its time source cluster; this pair of nodes is called a time gateway. The node on the time sink cluster side is then called a time gateway sink node while the node on the time source cluster side is called time gateway source node. Figure 1-3 depicts the basic setup. Depending on the synchronization method employed by time source cluster, the time gateway source node may be a TT-D coldstart node, a TT-D non-coldstart sync node, a TT-L coldstart node, or a non-sync node.[6] The two nodes of the time gateway are connected via a time gateway interface, which is used by the time gateway source node to provide information about the schedule of the time source cluster to the time gateway sink node.

Instead of the usual distributed startup, a TT-E coldstart node derives the cycle length and position of the cycle start from its time gateway source node and directly starts transmitting according to this schedule (slightly shifted with a fixed offset of *cdTSrcCycleOffset* microticks). Similar to the TT-L synchronization method, each TT-E coldstart node transmits two startup frames, so that a single TT-E coldstart node suffices to start and maintain a TT-E cluster. Contrary to the TT-L synchronization method, multiple TT-E coldstart nodes may be present in a TT-E cluster. As all TT-E coldstart nodes derive their schedule from the same cluster, they are implicitly synchronized to one another.

[6] In theory it is also possible for the time gateway source node to be a TT-E coldstart node, but this would imply a time gateway that includes three or more distinct nodes. Such configurations are beyond the scope of this specification.

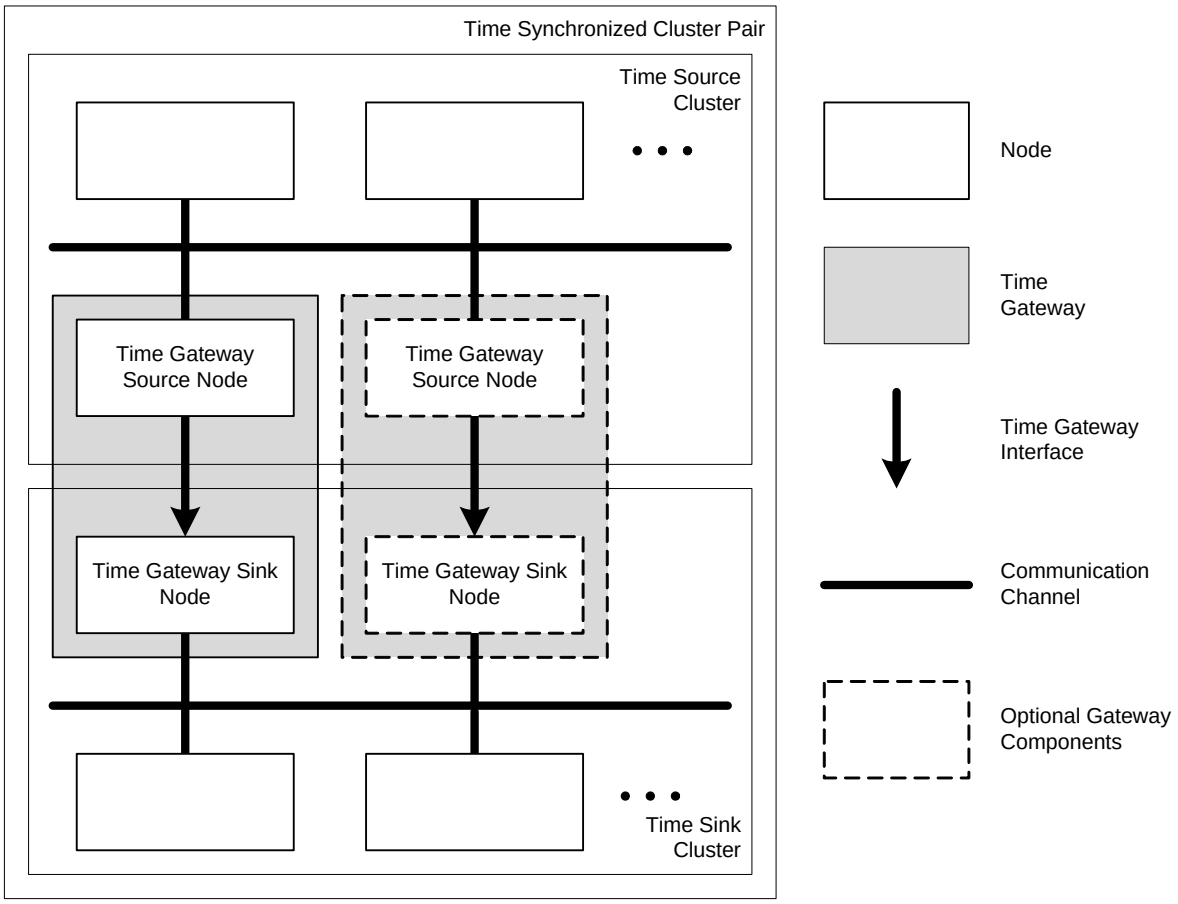

Figure 1-3: Time synchronized cluster pair.

The advantage of the TT-E synchronization method is the close coupling of the schedule of a TT-E cluster to another FlexRay cluster. In this way, a single FlexRay cluster may be split into synchronized sub-clusters to avoid limits on attached nodes placed upon a single FlexRay cluster by [EPL10] or to enable a separation of nodes into multiple clusters according to communication needs for a more efficient use of the available bandwidth.

Figure 1-4 shows the configuration of two connected clusters, the lower being a time sink cluster, the upper being a time source cluster, in this case a TT-D cluster. The TT-D cluster could also be replaced by a TT-L cluster or TT-E cluster.[7] The number of TT-E coldstart nodes i must be at least one and less than or equal to 7. The number of non-sync nodes k is not limited by the protocol.

The support of the TT-E synchronization method is optional. This means that a FlexRay node may not support being a TT-E coldstart node, may not support being a time gateway source node, or may support neither of these features.

[7] This specification does not provide configuration constraints for a "daisy-chain" of TT-E clusters. All configuration constraints assume that a time source cluster is either a TT-D cluster or a TT-L cluster.

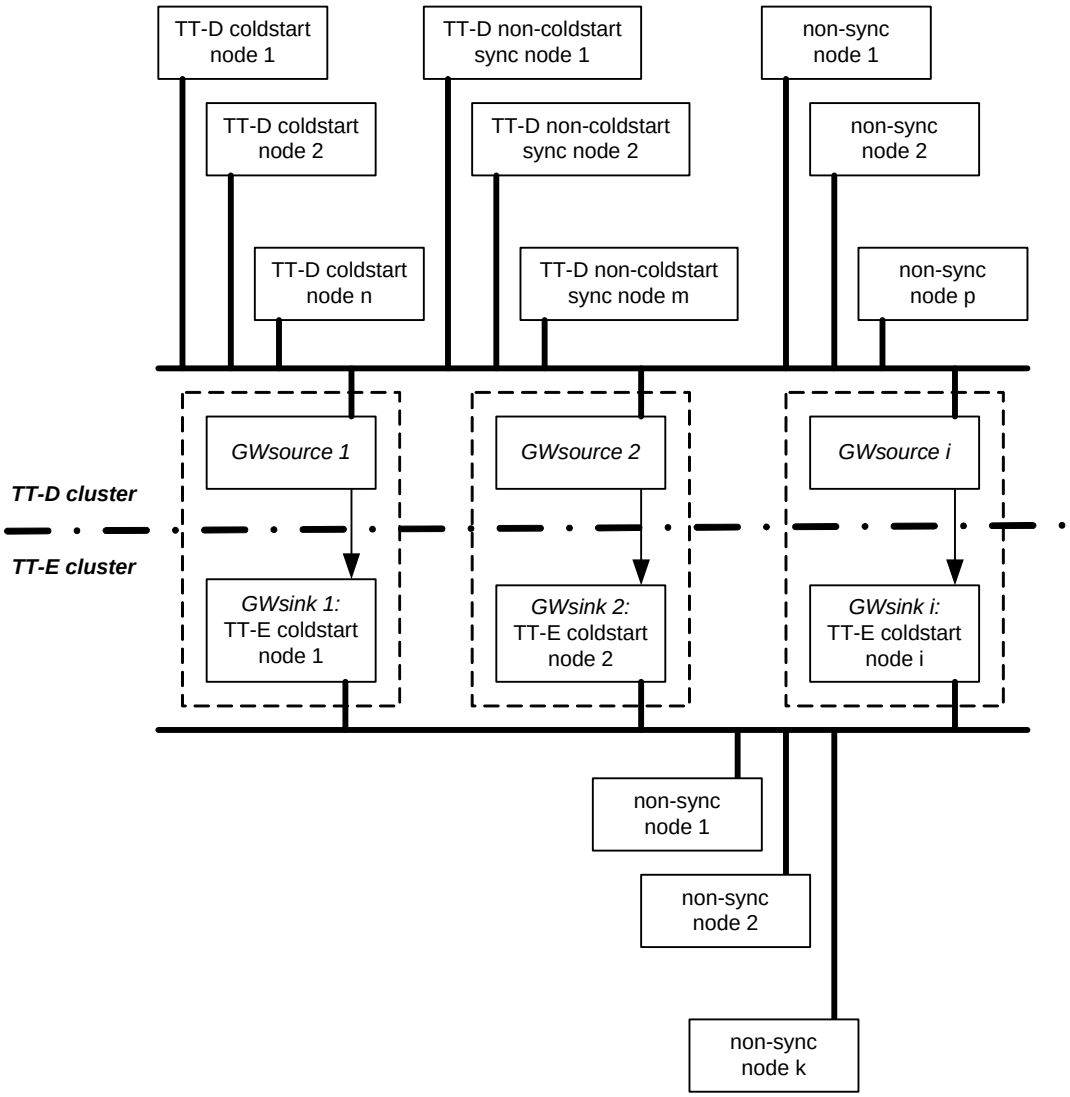

Figure 1-4: TT-E cluster.

Optional behavior related to the feature of being a TT-E coldstart node is marked by dashed rectangles within the SDL diagrams. Each such rectangle is additionally annotated with the text "TT-E time gateway sink behavior (optional)". An implementation not supporting the TT-E synchronization method may choose not to implement the SDL content marked as optional. Further, such an implementation shall behave as if the value of the variable *pExternalSync* and in consequence also of the variable *vExternalSync* is fixed to false.[8]

1.11 Network topology considerations

The following sections provide a brief overview of the possible topologies for a FlexRay system. This material is for reference only - detailed requirements and specifications may be found in [EPL10].

[8] As a result, it would be possible to redraw the SDL diagrams to remove the optional TT-E behavior by eliminating all decision boxes involving *pExternalSync* or *vExternalSync* (which under these circumstances have a pre-determined outcome) and all optional material inside the dashed boxes. The remainder is the required behavior for all FlexRay implementations.

There are several ways to design a FlexRay cluster. It can be configured as a single-channel or dual-channel bus network, a single-channel or dual-channel star network, or in various hybrid combinations of bus and star topologies.

A FlexRay cluster consists of at most two channels, identified as Channel A and Channel B. Each node in the cluster may be connected to either or both of the channels. In the fault free condition, all nodes connected to Channel A are able to communicate with all other nodes connected to Channel A, and all nodes connected to Channel B are able to communicate with all other nodes connected to Channel B. If a node needs to be connected to more than one cluster then the connection to each cluster must be made through a different communication controller[9].

1.11.1 Passive bus topology

Figure 1-5 shows the possible topology configuration of the communication network as a dual bus. A node can be connected to both channels A and B (nodes A, C, and E), only to channel A (node D), or only to channel B (node B).

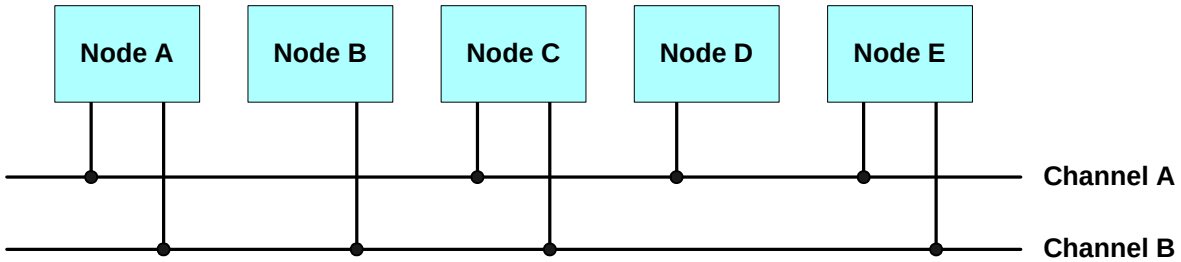

Figure 1-5: Dual channel bus configuration.

The FlexRay communication network can also be a single bus. In this case, all nodes are connected to this bus.

1.11.2 Active star topology

A FlexRay communication network can be built as a multiple star topology. Similar to the bus topology, the multiple-star topology can support redundant communication channels. Each network channel must be free of closed rings, and there can be no more than two active stars on a network channel[10]. Note that there may be physical layer-related restrictions that limit the number of active stars for certain bit rates - see [EPLAN10] for details.

An incoming signal received on a branch of an active star is actively driven to all other branches of the active star.

[9] For example, it is not allowed for a communication controller to connect to Channel A of one cluster and Channel B of another cluster.

[10] A channel with two active stars would have the stars connected to each other. Communication between nodes connected to different stars would pass through both stars (a cascaded star topology).

The configuration of a single redundant star network is shown in Figure 1-6. The logical structure (i.e., the node connectivity) of this topology is identical with that shown in Figure 1-5. It is also possible to create a single, non-redundant star topology that has the same logical structure as the single bus mentioned above.

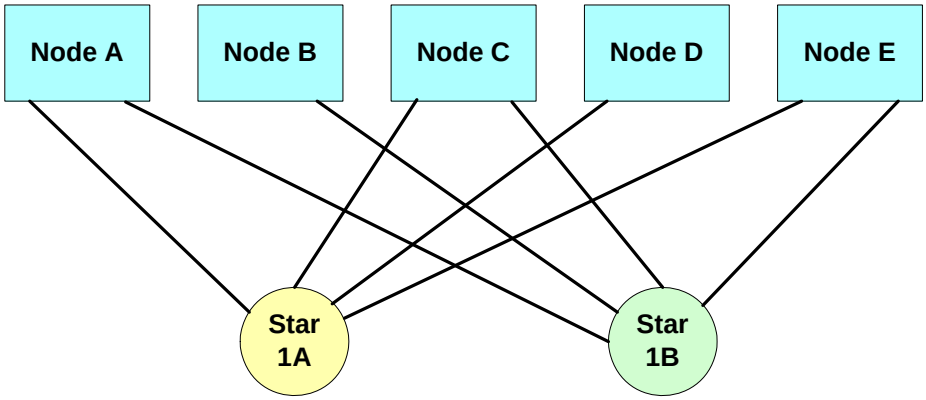

Figure 1-6: Dual channel single star configuration.

Figure 1-7 shows a single channel network built with two active stars. Each node has a point-to-point-connection to one of the two active stars. The first active star (1A) is directly connected to the second active star (2A).

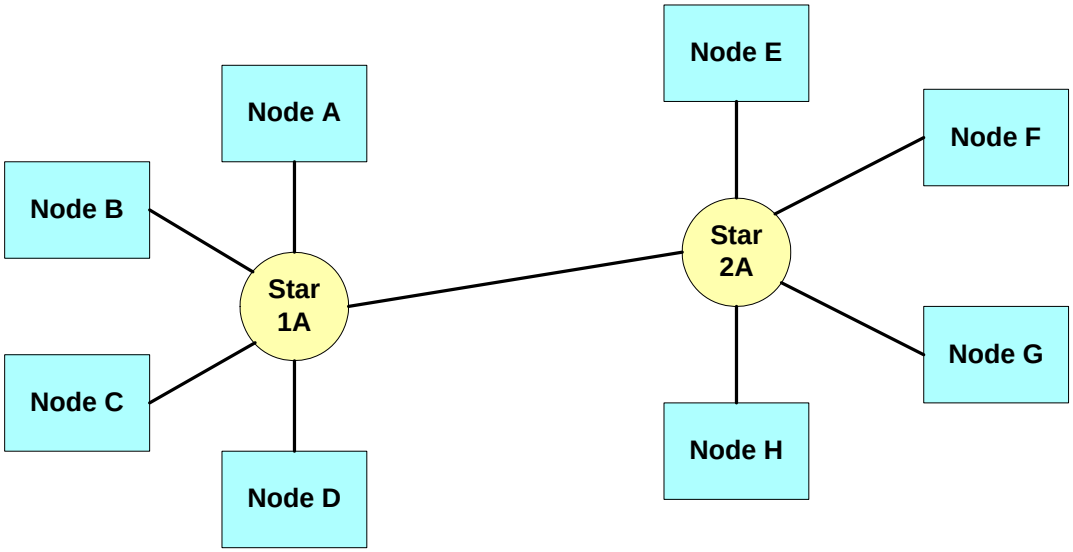

Figure 1-7: Single channel cascaded star configuration.

Note that it is also possible to have a redundant channel configuration with cascaded stars. An example of such a configuration is Figure 1-8. Note that this example does not simply replicate the set of stars for the second channel - Star 1A connects nodes A, B, and C, while Star 1B connects nodes A, C, and E.

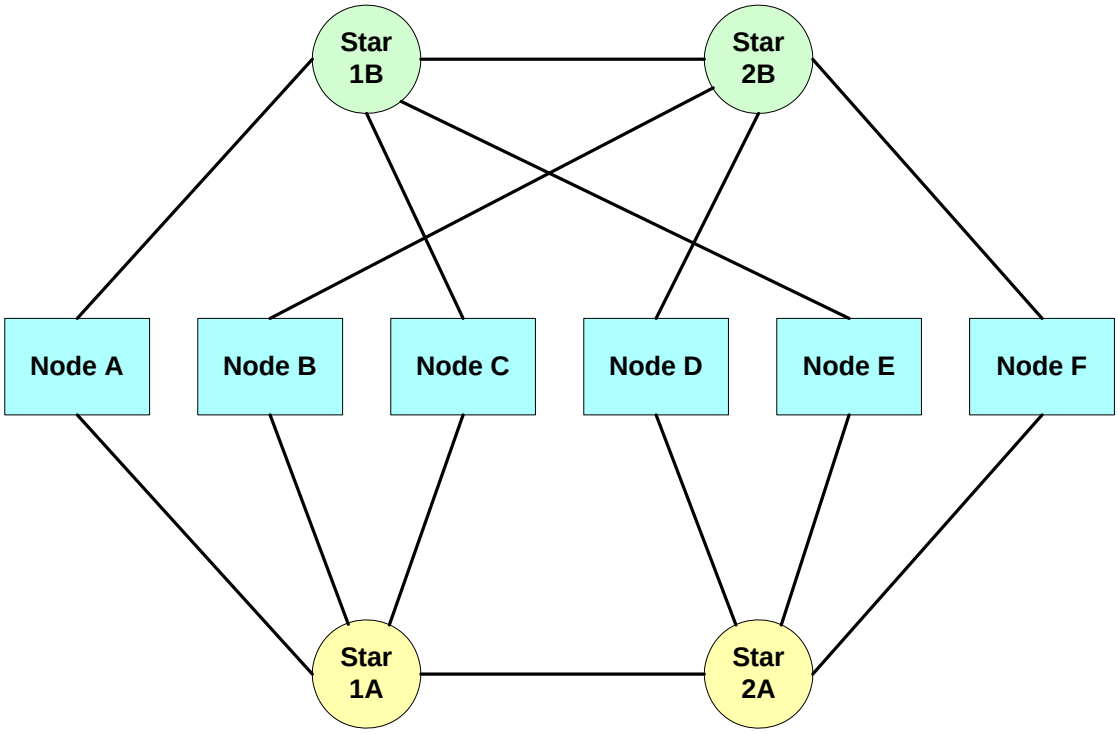

Figure 1-8: Dual channel cascaded star configuration.

1.11.3 Active star topology combined with a passive bus topology

In addition to topologies that are composed either entirely of a bus topology or entirely of a star topology, it is possible to have hybrid topologies that are a mixture of bus and star configurations. The FlexRay system supports such hybrid topologies as long as the limits applicable to each individual topology are not exceeded. For example, the limit of two cascaded active stars also limits the number of cascaded active stars in a hybrid topology.

There are a large number of possible hybrid topologies, but only two representative topologies are shown here. Figure 1-9 shows an example of one type of hybrid topology. In this example, some nodes (nodes A, B, C, and D) are connected using point-to-point connections to an active star. Other nodes (nodes E, F, and G) are connected to each other using a bus topology. This bus is also connected to an active star, allowing nodes E, F, and G to communicate with the other nodes.

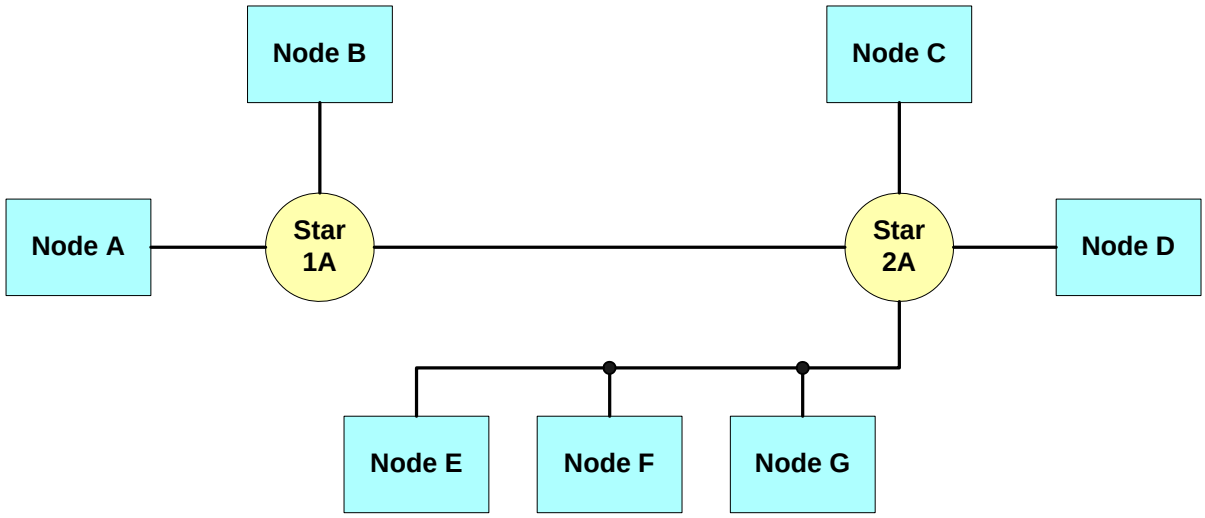

Figure 1-9: Single channel hybrid example.

A fundamentally different type of hybrid topology is shown in Figure 1-10. In this case, different topologies are used on different channels. Here, channel A is implemented as a bus topology connection, while channel B is implemented as a star topology connection.

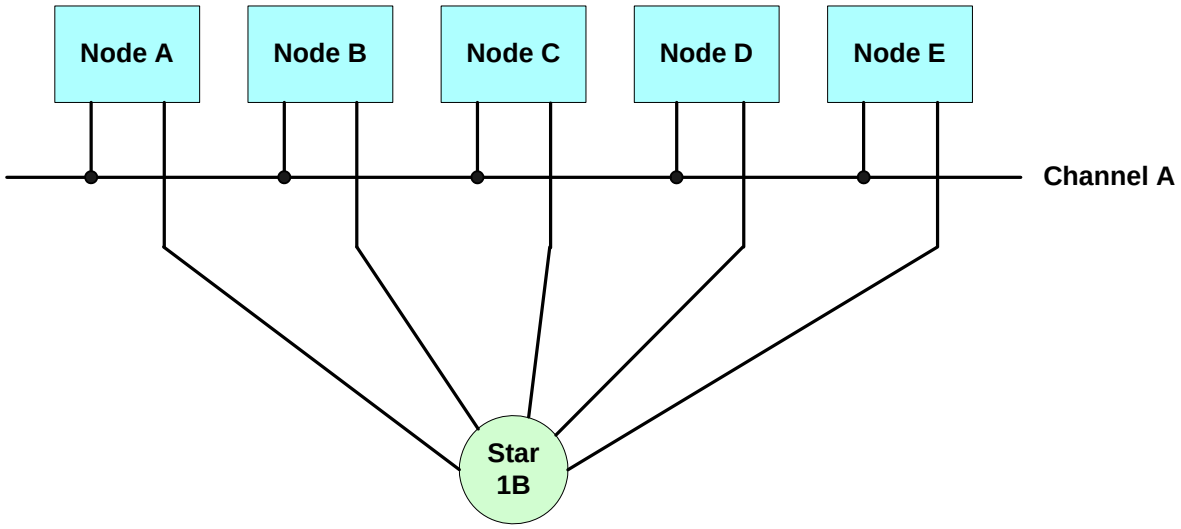

Figure 1-10: Dual channel hybrid example.

The protocol implications of topologies with stubs on the connection between active stars have not been fully analyzed. As a result, such topologies are not recommended and are not considered in this specification.

1.12 Example node architecture

1.12.1 Objective

This section is intended to provide insight into the FlexRay architecture by discussing an example node architecture and the interfaces between the FlexRay hardware devices.

The information in this section is for reference only. The detailed specification of the interfaces is given in the electrical physical layer specification [EPL10]; references are made here to appropriate text passages from this document.

Note that an active star component can also function in a role similar to a bus driver via the use of the optional BD-CC interface as described in [EPL10]. The following sections describe the node architecture under the assumption that the CC interfaces to the channel(s) via a bus driver rather than via an active star.

1.12.2 Overview

Figure 1-11 depicts an example node architecture. One communication controller, one host, one power supply unit, and two bus drivers are depicted. Each communication channel has one bus driver to connect the node to the channel. In addition to the indicated communication and control data interfaces an optional interface between the bus driver and the power supply unit may exist.

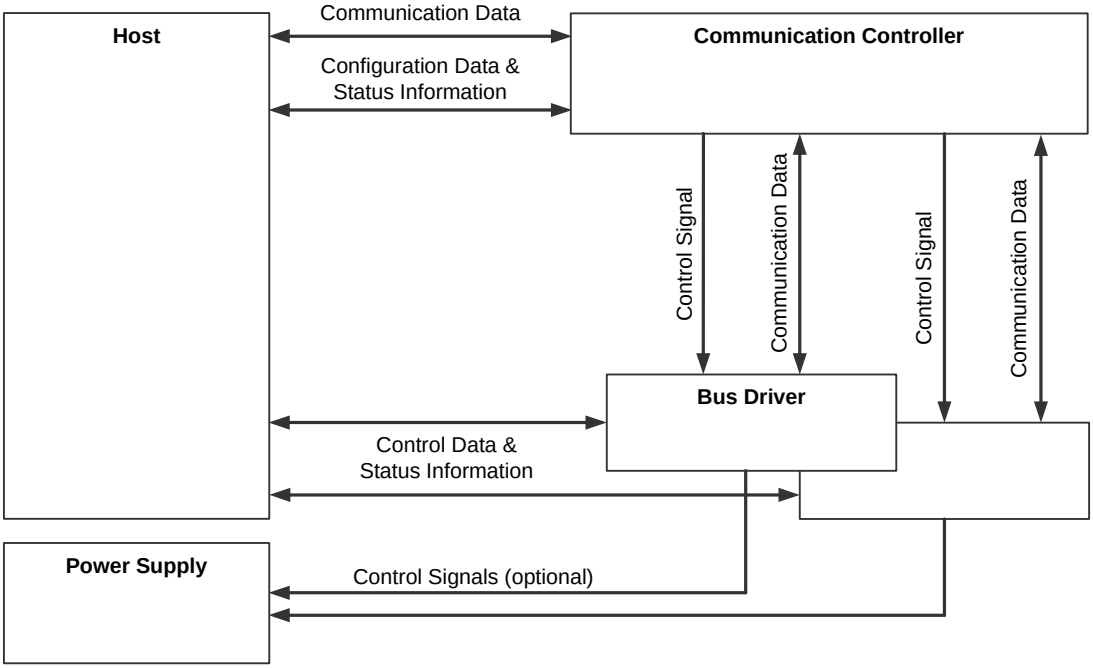

Figure 1-11: Logical interfaces.

1.12.3 Host - communication controller interface

The host and the communication controller share a substantial amount of information. The host provides control and configuration information to the CC, and provides payload data that is transmitted during the communication cycle. The CC provides status information to the host and delivers payload data received from communication frames.

Details of the interface between the host and the communication controller are specified in Chapter 9.

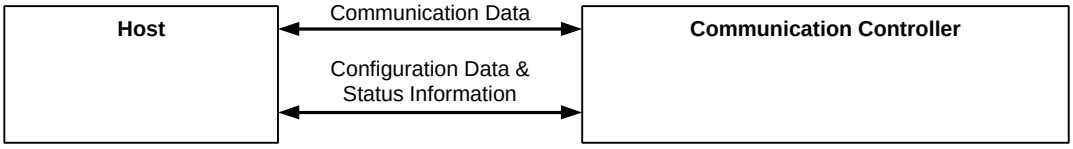

Figure 1-12: Host - communication controller interfaces.

1.12.4 Communication controller - bus driver interface

The interface between the BD and the CC consists of three digital electrical signals. Two are outputs from the CC (TxD and TxEN) and one is an output from the BD (RxD).

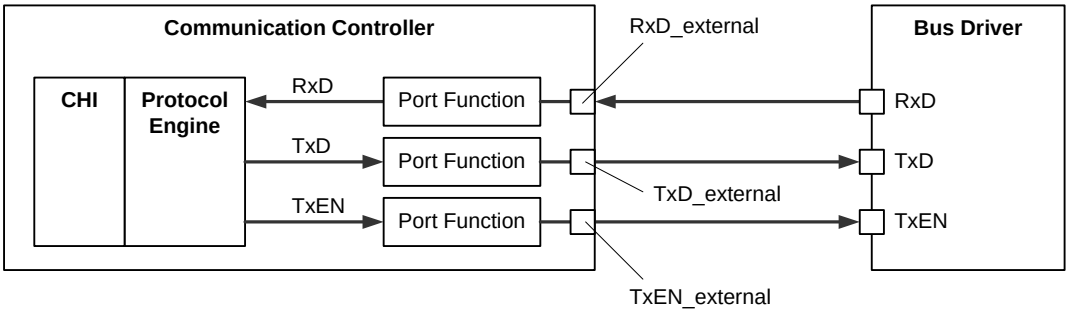

Figure 1-13: Communication controller - bus driver interface.

The CC uses the TxD (Transmit Data) signal to transfer the actual signal sequence to the BD for transmission onto the communication channel. TxEN (Transmit Data Enable Not) indicates the CC's request to have the bus driver present the data on the TxD line to its corresponding channel.

The BD uses the RxD (Receive Data) signal to transfer the actual received signal sequence to the CC.

Figure 1-13 shows the connection between the communication controller and the bus driver and the internal connection between the protocol engine and the pins. This protocol specification does not specify a device but the behavior of the FlexRay protocol. In the following the protocol specification only refers to the internal signals TxD, TxEN, and RxD as depicted in Figure 1-13.

Between the internal and the external signals there are device specific port functions which are responsible for the electrical behavior of the pins, for example:

- I/O voltage level,
- ESD protection,
- behavior during power up initialization, reset, or while depowered,
- pin multiplexing (e.g. the connection of the external pins associated with the RxD_external, TxD_external and TxEN_external signals either to the FlexRay protocol engine (i.e., the RxD, TxD, and TxEN signals, respectively), to some other function inside the CC implementation[11], or to nothing at all).

If the pins are connected to something other than the FlexRay protocol engine this specification places no requirements on the behavior of those pins. However, the behavior during power up initialization, reset, while depowered, and the default behavior prior to the configuration of any pin multiplexing, shall ensure that the bus driver does not actively drive the FlexRay bus[12], and that the bus driver interprets the TxD signal as low[13].

[11] For example, if the CC implementation is part of a microcontroller it is possible that the microcontroller could be configured to use I/O pins either as the FlexRay I/O (RxD, TxD, and TxEN) or for some other purpose (perhaps general purpose I/O).

Some requirements (e.g. the electrical characteristics and timing) on the port functions and the TxD_external, TxEN_external and RxD_external signals are specified in [EPL10].

1.12.5 Bus driver - host interface

The interface between the BD and the host allows the host to control the operating modes of the BD and to read error conditions and status information from the BD.

This interface can be realized using hard-wired signals (see option A in Figure 1-14) or by a Serial Peripheral Interface (SPI) (see option B in Figure 1-15).

1.12.5.1 Hard wired signals (option A)

This implementation of the BD - host interface uses discrete hard wired signals. The interface consists of at least an STBN (Standby Not) signal that is used to control the BD's operating mode and an ERRN (Error Not) signal that is used by the BD to indicate detected errors. The interface could also include additional control signals (the "EN" signal is shown as an example) that support control of optional operational modes.

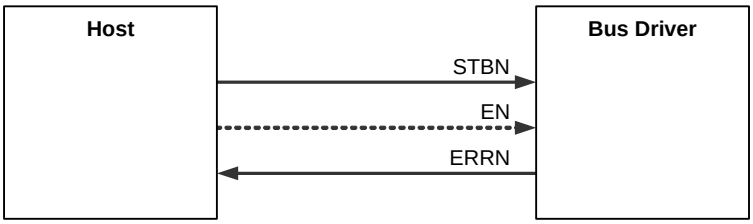

Figure 1-14: Example bus driver - host interface (option A).

This interface is product specific; some restrictions are given in [EPL10] that define minimum functionality to ensure interoperability.

1.12.5.2 Serial peripheral interface (SPI) (option B)

This implementation of the BD - host interface uses an SPI link to allow the host to command the BD operating mode and to read out the status of the BD. In addition, the BD has a hardwired interrupt output (INTN).

The electrical characteristics and timing of this interface are specified in [EPL10].

[12] This could be done, for example, by ensuring that the TxEN_external output is driven to active high, provided with a weak pull up, or set to high impedance.

[13] This could be done, for example, by ensuring that the TxD_external output is driven to active low, provided with a weak pull down, or set to high impedance.

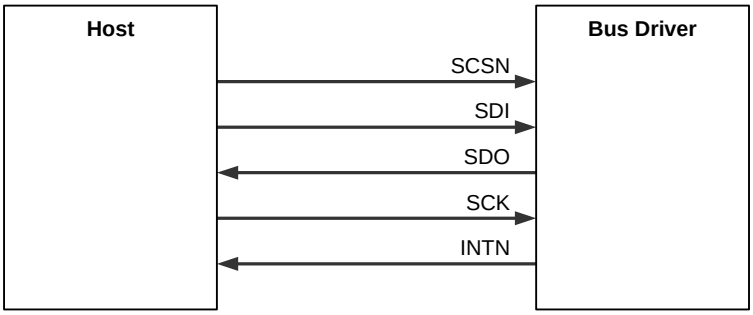

Figure 1-15: Example bus driver - host interface (option B).

1.12.6 Bus driver - power supply interface (optional)

The inhibit signal (INH1) is an optional interface that allows the BD to directly control the power supply of an ECU. This signal could also be used as one of a set of signals that control the power moding of the ECU.

Figure 1-16: Bus driver - power supply interface.

The electrical characteristics and behavior of the INH1 signal are specified in [EPL10].

1.12.7 Time gateway interface

A time gateway sink node needs information on the schedule and clock synchronization algorithm of its time gateway source node. The time gateway source node provides this information via the time gateway interface to the time gateway sink node. This interface is unidirectional - no information flows back from the time gateway sink node to the time gateway source node. This interface is an optional feature only required to allow the node to be a time gateway source or time gateway sink node. Details of the interface between the time gateway source node and the time gateway sink node are specified in Chapter 8. The usage of this interface is indicated in the SDL diagrams by the EXP keyword on the transmitting end and the IMP keyword on the receiving end.

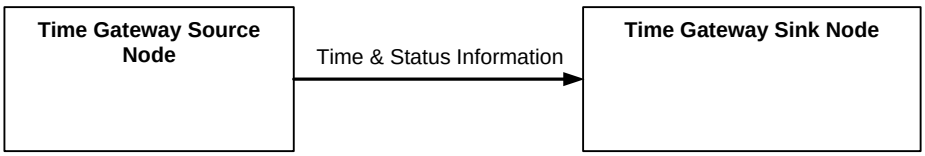

Figure 1-17: Time gateway interface.

1.13 Testability requirements

The FlexRay Protocol Conformance Test Specification [PCT10] contains additional implementation requirements. The purpose of these requirements is to facilitate testing, for example by establishing timing bounds for the availability of CHI information necessary to execute certain tests.

Chapter 2
Protocol Operation Control

This chapter defines how the core mechanisms of the protocol are moded in response to host commands and protocol conditions.

2.1 Principles

The primary protocol behavior of FlexRay is embodied in four core mechanisms, each of which is described in a dedicated chapter of this specification.

- Coding and Decoding (see Chapter 3)
- Media Access Control (see Chapter 5)
- Frame and Symbol Processing (see Chapter 6)
- Clock Synchronization (see Chapter 8)

In addition, the controller host interface (CHI) provides the mechanism for the host to interact in a structured manner with these core mechanisms and for the protocol mechanisms, including Protocol Operation Control (POC), to provide feedback to the host (see Chapter 9).

Each of the core mechanisms possesses modal behavior that allows it to alter its fundamental operation in response to high-level mode changes of the node. Proper protocol behavior can only occur if the mode changes of the core mechanisms are properly coordinated and synchronized. The purpose of the POC is to react to host commands and protocol conditions by triggering coherent changes to core mechanisms in a synchronous manner, and to provide the host with the appropriate status regarding these changes.

The necessary synchronization of the core mechanisms is particularly evident during the wakeup, startup and reintegration procedures. These procedures are described in detail in Chapter 7. However, these procedures are wholly included in the POC as macros in the POC SDL models. They can be viewed as specialized extensions of the POC.

2.1.1 Communication controller power moding

Before the POC can perform its prescribed tasks the communication controller (CC) must achieve a state where there is a stable power supply. Furthermore, the POC can only continue to operate while a stable power supply is present.

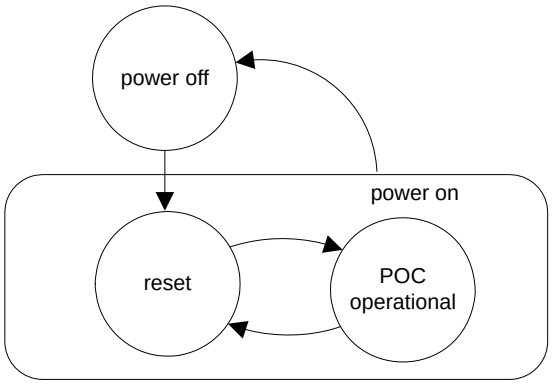

Figure 2-1: Power moding of the communication controller.

Figure 2-1 depicts an overview of the CC power moding operation. Note that in this figure the state labeled *POC operational* is not actually a specific state but rather a superset of all operational states of the Protocol Operation Control (see Figure 2-3).

In the *power off* state there is insufficient power for the CC to operate[14]. In the *power on* state (including both *reset* and *POC operational*) the CC shall guarantee that all pins are in prescribed states. In the POC operational state the CC shall drive the pins in accordance with the product specification. The POC controls the other protocol mechanisms in the manner described in this chapter while the CC is in the *POC operational* state.

2.2 Description

The relationships between the CHI, POC and the core mechanisms are depicted in Figure 2-2[15].

[14] While the CC cannot enforce specific behavior of the pins, there shall be product-specific behavior specified (e.g. high impedance).

[15] The dark lines represent data flows between mechanisms that are relevant to this chapter. The lighter gray lines are relevant to the protocol, but not to this chapter.

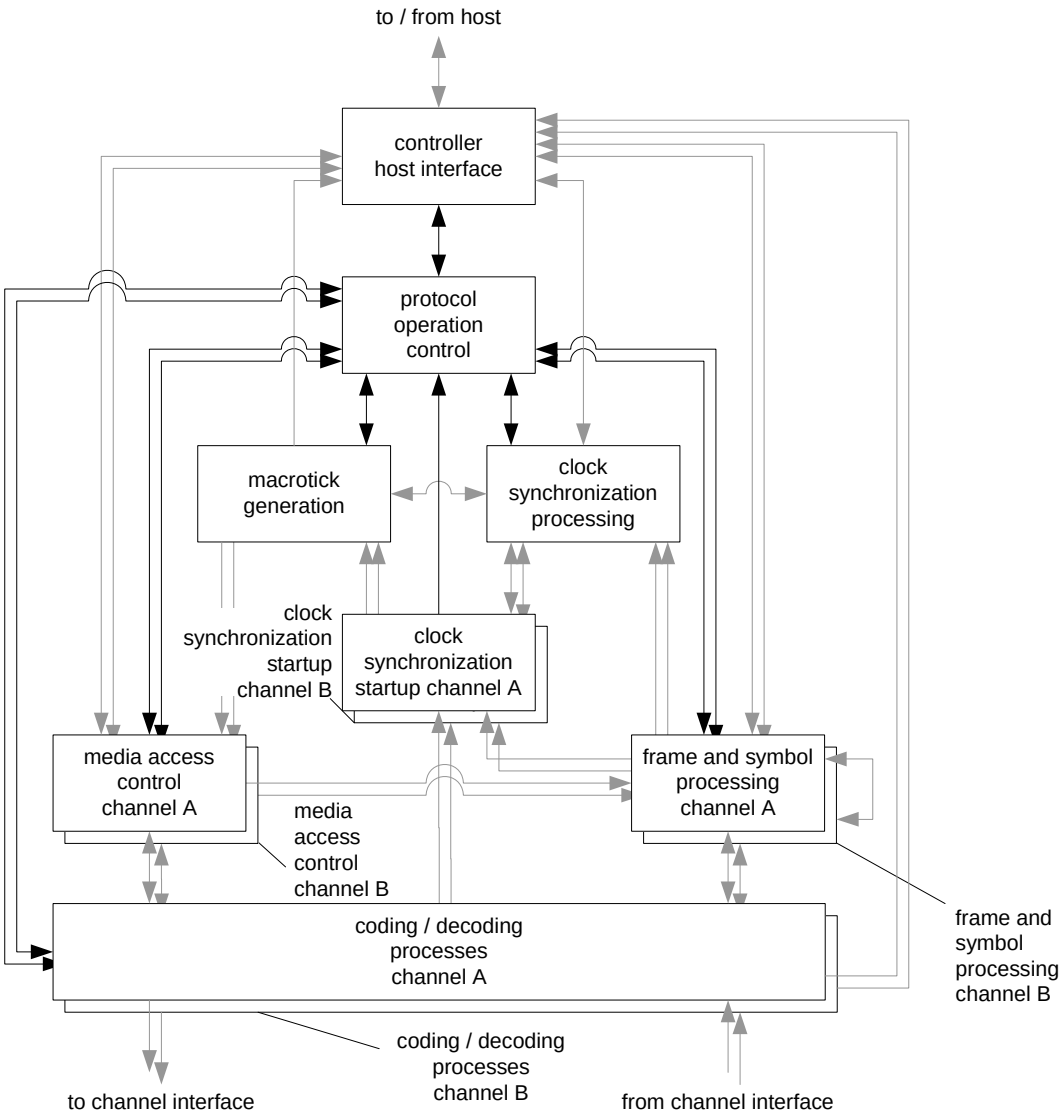

Figure 2-2: Protocol operation control context.

2.2.1 Operational overview

The POC SDL process is created as the CC enters the *POC operational* state and terminated when the CC exits it. The POC process is responsible for creating the SDL processes corresponding to the core mechanisms and informing those processes when they are required to terminate. It is also responsible for changing the mode of the core mechanisms of the protocol in response to changing conditions in the node.

Mode changes of the core mechanisms occur when the POC itself changes states. Some of the POC state changes are simply a consequence of completing tasks. For example, the *POC:normal active* state (see section 2.3.6) is entered as a consequence of completing the startup process. However, most of the POC state changes are a direct consequence of one of the following:

- Host commands communicated to the POC via the CHI.
- Error conditions detected either by the protocol engine or a product-specific built-in self-test (BIST) or sanity check. The host may also perform sanity checks, but the consequences of the host sanity checks are indicated to the POC as host commands.

2.2.1.1 Host commands

Strictly speaking, the POC is unaware of the commands issued by the host. Host interactions with the CC are processed by the CHI. The CHI is responsible for relaying relevant commands to the POC. While this is a minor distinction, the remainder of the POC description in this document treats the host commands as if they originated in the CHI. Similarly, status information from the POC that is intended for the host is simply provided to the CHI, which is then responsible for formatting it appropriately and relaying it to the host in a prescribed manner (see Chapter 9).

Some host commands result in immediate changes being reflected in the moding of the core mechanisms while mode changes are deferred to the end of the communication cycle for others. In addition, some host commands are not processed in every POC state. The detailed behavior corresponding to each command is captured in the SDL descriptions and accompanying text (see section 2.3). They are briefly summarized in Table 2-1. If the host issues a specific CHI command while the POC is in a state other than the states shown in the "Where processed (POC States)" column in Table 2-1 the command shall be ignored (i.e., it shall have no effect on the protocol engine).

CHI command	Where processed (POC States)	When processed
ALL_SLOTS	*POC:normal active*, *POC:normal passive*	End of cycle
ALLOW_COLDSTART	All except *POC:default config*, *POC:config*, *POC:halt*, *POC:wakeup listen*, *POC:wakeup send*, *POC:wakeup detect*[a]	Immediate
CLEAR_DEFERRED	All except *POC:default config*, *POC:config*, *POC:ready*, *POC:halt*	Immediate
CONFIG	*POC:default config*, *POC:ready*	Immediate
CONFIG_COMPLETE	*POC:config*	Immediate
DEFAULT_CONFIG	*POC:halt*	Immediate
RUN	*POC:ready*	Immediate
WAKEUP	*POC:ready*	Immediate
FREEZE	All	Immediate
IMMEDIATE_READY	All except *POC:default config*, *POC:config*, *POC:ready*, *POC:halt*	Immediate
DEFERRED_READY	All except *POC:default config*, *POC:config*, *POC:ready*, *POC:halt*, *POC:normal active*, *POC:normal passive*, *POC:wakeup send*, *POC:coldstart collision resolution*, *POC:cold-start consistency check*, *POC:coldstart join*	Immediate
DEFERRED_READY	*POC:normal active*, *POC:normal passive*, *POC:coldstart collision resolution*[b], *POC:cold-start consistency check*, *POC:coldstart join*	End of cycle
DEFERRED_READY	*POC:wakeup send*	After transmission of a complete WUP or detection of wakeup collision

CHI command	Where processed (POC States)	When processed
DEFERRED_HALT	All except *POC:halt*, *POC:normal active*, *POC:normal passive*, *POC:wakeup send*, *POC:coldstart collision resolution*, *POC:cold-start consistency check*, *POC:coldstart join*	Immediate
DEFERRED_HALT	*POC:normal active*, *POC:normal passive*, *POC:coldstart collision resolution*[b], *POC:cold-start consistency check*, *POC:coldstart join*	End of cycle
DEFERRED_HALT	*POC:wakeup send*	After transmission of a complete WUP or detection of wakeup collision

a. The ALLOW_COLDSTART command is processed as described in Figure 2-13 except when the POC is in the *POC:integration listen* state, in which case it is processed by the SDL in Figure 7-22.
b. In the *POC:coldstart collision resolution* state a deferred command is either processed at the end of the cycle or after a frame header or a CAS is received.

Table 2-1: CHI host command summary.

2.2.1.2 Error conditions

The POC contains two basic mechanisms for responding to errors. For significant errors, the *POC:halt* state is immediately entered. The POC also contains a three-state degradation model for errors that can be endured for a limited period of time. In this case entry to the *POC:halt* state is deferred, at least temporarily, to support possible recovery from a potentially transient condition.

2.2.1.2.1 Errors causing immediate entry to the *POC:halt* state

There are three general conditions that trigger entry to the *POC:halt* state:

- Product-specific error conditions such as BIST errors and sanity checks.
- Error conditions detected by the host that result in a FREEZE command being sent to the POC via the CHI.
- Fatal error conditions detected by the FSP process.

Product-specific errors are accommodated by the POC, but not described in this specification (see section 2.3.3). Similarly, host detected error strategies are supported by the POC's ability to respond to a host FREEZE command (see section 2.3.3), but the host-based mechanisms that trigger the command are beyond the scope of this specification. Only errors detected by the POC or one of the core mechanisms are explicitly detailed in this specification.

2.2.1.2.2 Errors handled by the degradation model

Integral to the POC is a three-state error handling mechanism referred to as the degradation model. It is designed to react to certain conditions detected by the clock synchronization mechanism that are indicative of a problem, but that may not require immediate action due to the inherent fault tolerance of the clock synchronization mechanism. This makes it possible to avoid immediate transitions to the *POC:halt* state while assessing the nature and extent of the errors.

The degradation model is embodied in three POC states - *POC:normal active*, *POC:normal passive*, and *POC:halt*.

In the *POC:normal active* state the node is assumed to be either error free, or at least within error bounds that allow continued "normal operation". Specifically, it is assumed that the node remains adequately time-synchronized to the cluster to allow continued frame transmission without disrupting the transmissions of other nodes.

In the *POC:normal passive* state, it is assumed that synchronization with the remainder of the cluster has degraded to the extent that continued frame transmissions cannot be allowed because collisions with transmissions from other nodes are possible. Frame reception continues in the *POC:normal passive* state in support of host functionality and in an effort to regain sufficient synchronization to allow a transition back to the *POC:normal active* state.

If errors persist in the *POC:normal passive* state or if errors are severe enough, the POC can transition to the *POC:halt* state. In this state it is assumed that recovery back to the *POC:normal active* state cannot be achieved, so the POC halts the core mechanisms in preparation for reinitializing the node.

The conditions for transitioning between the three states comprising the degradation model are configurable. Furthermore, transitions between the states are communicated to the host allowing the host to react appropriately and to possibly take alternative actions using one of the explicit host commands.

2.2.1.3 POC status

In order for the host to react to POC state changes, the host must be informed when POC state changes occur. This is the responsibility of the CHI. The POC supports the CHI by providing appropriate information to the CHI.

The basic POC status information is provided to the CHI using the *vPOC* data structure. *vPOC* is of type *T_POCStatus*, which is defined in Definition 2-1:

```
newtype T_POCStatus
struct
      State             T_POCState;
      Freeze            Boolean;
      CHIHaltRequest    Boolean;
      CHIReadyRequest   Boolean;
      ColdstartNoise    Boolean;
      SlotMode          T_SlotMode;
      ErrorMode         T_ErrorMode;
      WakeupStatus      T_WakeupStatus;
      StartupState      T_StartupState;
endnewtype;
```

Definition 2-1: Formal definition of T_POCStatus.

The *vPOC* structure is an aggregation of nine distinct status variables.

vPOC!State is used to indicate the state of the POC and is based on the *T_POCState* formal definition in Definition 2-2.

```
newtype T_POCState
      literals CONFIG, DEFAULT_CONFIG, HALT, NORMAL_ACTIVE, NORMAL_PASSIVE,
      READY, STARTUP, WAKEUP;
endnewtype;
```

Definition 2-2: Formal definition of T_POCState.

vPOC!Freeze is used to indicate that the POC has entered the *POC:halt* state due to an error condition requiring an immediate halt (see section 2.3.3). *vPOC!Freeze* is Boolean.

vPOC!CHIHaltRequest is used to indicate that a request has been received from the CHI to halt the POC at the end of the communication cycle (see section 2.3.4.1). *vPOC!CHIHaltRequest* is Boolean.

vPOC!CHIReadyRequest is used to indicate that a request has been received from the CHI to enter the *POC:ready* state at the end of the communication cycle (see section 2.3.4.1). *vPOC!CHIReadyRequest* is Boolean.

vPOC!ColdstartNoise indicates noisy channel conditions during *POC:coldstart listen* if the coldstart attempt of a leading coldstart node was completed successfully (see section 7.2). *vPOC!ColdstartNoise* is Boolean.

vPOC!SlotMode is used to indicate what slot mode the POC is in (see sections 2.3.4.2, 2.3.7.1.2, and 2.3.7.1.3). *vPOC!SlotMode* is based on the *T_SlotMode* formal definition in Definition 2-3.

```
newtype T_SlotMode
      literals KEYSLOT, ALL_PENDING, ALL;
endnewtype;
```

Definition 2-3: Formal definition of T_SlotMode.

vPOC!ErrorMode is used to indicate what error mode the POC is in (see sections 2.3.7.1.2 and 2.3.7.1.3). *vPOC!ErrorMode* is based on the *T_ErrorMode* formal definition in Definition 2-4.

```
newtype T_ErrorMode
      literals ACTIVE, PASSIVE, COMM_HALT;
endnewtype;
```

Definition 2-4: Formal definition of T_ErrorMode.

vPOC!WakeupStatus is used to indicate the outcome of the execution of the WAKEUP mechanism (see Figure 2-12 and section 7.1.3.1). *vPOC!WakeupStatus* is based on the *T_WakeupStatus* formal definition in Definition 2-5.

```
newtype T_WakeupStatus
      literals UNDEFINED, RECEIVED_HEADER, RECEIVED_WUP, COLLISION_HEADER,
      COLLISION_WUP, COLLISION_UNKNOWN, TRANSMITTED;
endnewtype;
```

Definition 2-5: Formal definition of T_WakeupStatus.

The meaning of the individual *T_WakeupStatus* values is outlined in section 7.1.3.1.

vPOC!StartupState is used to indicate the current substate of the startup procedure (see section 7.2.4). *vPOC!StartupState* is based on the *T_StartupState* formal definition in Definition 2-6.

```
newtype T_StartupState
      literals UNDEFINED, COLDSTART_LISTEN, INTEGRATION_COLDSTART_CHECK,
      COLDSTART_JOIN, COLDSTART_COLLISION_RESOLUTION,
      COLDSTART_CONSISTENCY_CHECK, INTEGRATION_LISTEN, INITIALIZE_SCHEDULE,
      INTEGRATION_CONSISTENCY_CHECK, COLDSTART_GAP, EXTERNAL_STARTUP;
endnewtype;
```

Definition 2-6: Formal definition of T_StartupState.

The individual *T_StartupState* values are the states within the STARTUP mechanism in section 7.2.4.

In addition to the *vPOC* data structure, the POC makes two counters available to the host via the CHI. These counters are *vClockCorrectionFailed* and *vAllowPassiveToActive*, and are described in section 2.3.7.1.4.

2.2.1.4 SDL considerations for single channel nodes

FlexRay supports configurations where a node is only attached to one of the two possible FlexRay channels (see section 1.11).

Process instantiation is depicted in Figure 2-5. The channel specific processes are readily identifiable by the "_A" or "_B" text in the process names. The POC only instantiates the channel specific processes related to channels that are actually attached.

Process termination signal generation is also depicted in Figure 2-5. The channel specific signals are identifiable by the "_A" or "_B" text in the signal names. Termination signals sent to a channel specific process that is not instantiated will have no effect.

Process moding signals are generated throughout the POC. Figure 2-4 is an example that includes all of these moding signals. The channel specific moding signals are identifiable by the "on A" or "on B" text in the signal names. Moding signals, or any other signals, sent to a channel specific process that is not instantiated will have no effect.

2.3 The protocol operation control process

This section contains the formalized specification of the POC process. Figure 2-3 depicts an overview of the POC states and how they interrelate[16].

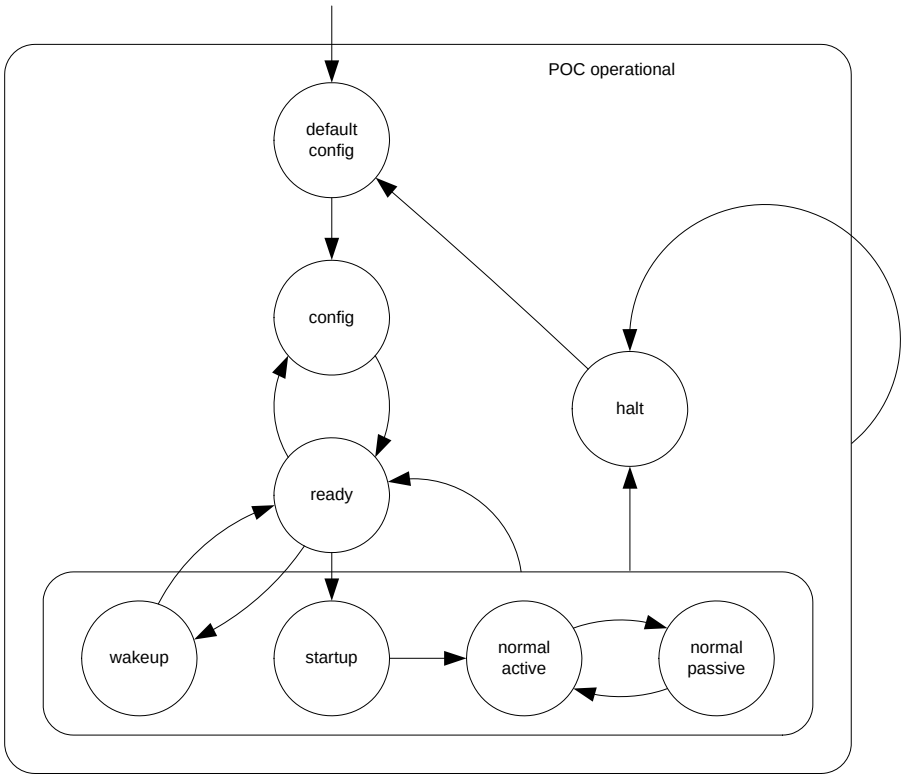

Figure 2-3: Overview of protocol operation control.

2.3.1 POC SDL utilities

The nature of the POC is that it performs tasks that often influence all of the core mechanisms simultaneously. From the perspective of SDL depiction these tasks are visually cumbersome. Macros are used in the POC for the sole purpose of simplifying the SDL presentation.

In the SDL that follows, there are several instances where the POC transitions to the *POC:ready* or *POC:halt* states. Prior to doing so, the core mechanisms have to be moded appropriately. The two macros in Figure 2-4 perform these tasks. PROTOCOL_ENGINE_READY modes the core mechanisms appropri-

[16] The states depicted as wakeup and startup are actually procedures containing several states. The depiction is simplified for the purpose of an overview.

ately for entry to *POC:ready*, and PROTOCOL_ENGINE_HALT modes the core mechanisms appropriately for entry to *POC:halt*.

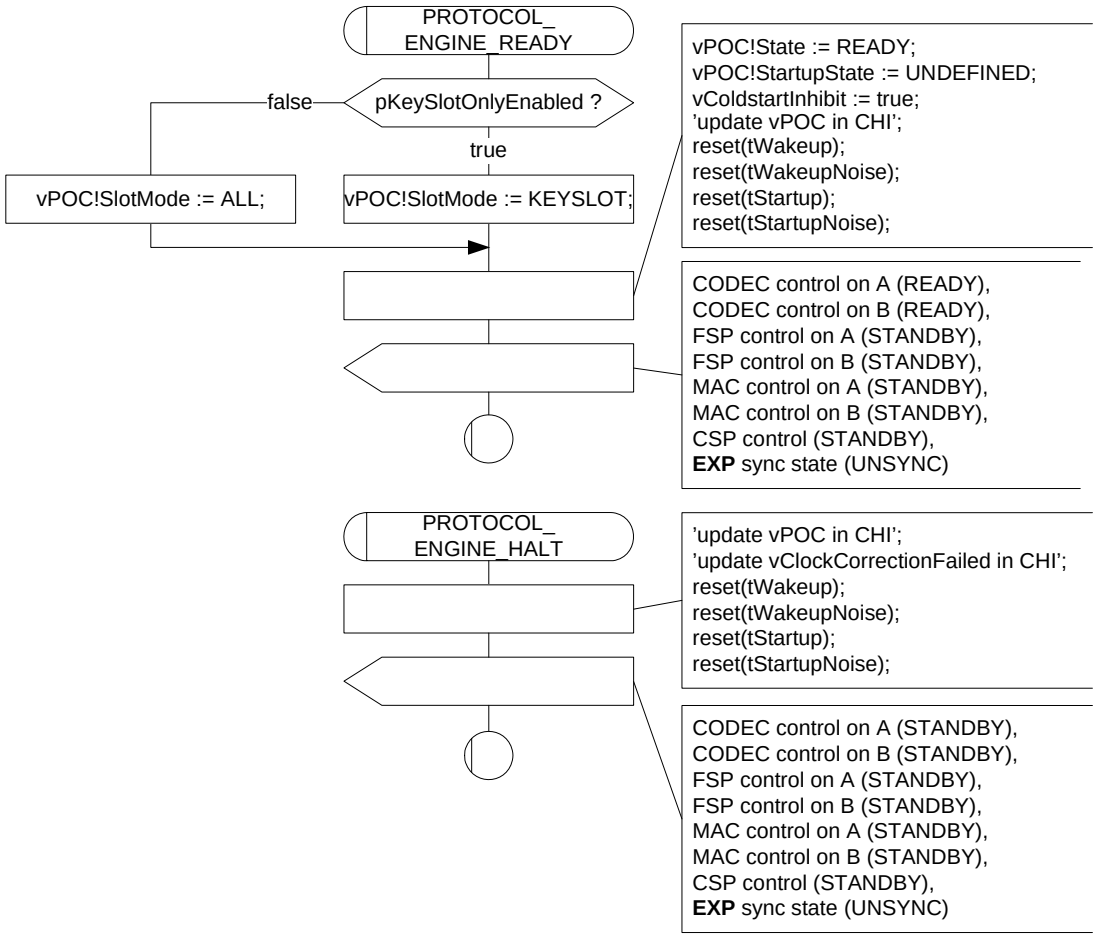

Figure 2-4: Macros to mode the core mechanisms for transitions to the *POC:ready* and *POC:halt* states [POC].

The SDL processes associated with the core mechanisms are created simultaneously by the POC. While the processes must terminate themselves, the POC is also responsible for simultaneously triggering this in all of the processes. Figure 2-5 depicts the macros for performing these two tasks.

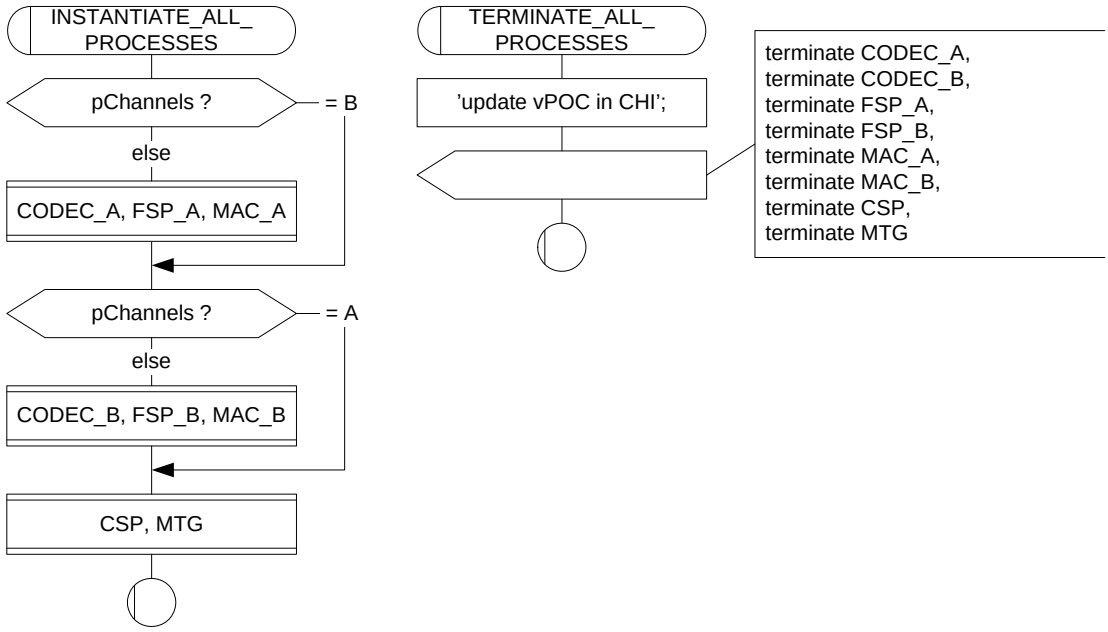

Figure 2-5: Macros for creating and terminating processes [POC].

2.3.2 SDL organization

From the perspective of procedural flow, the behavior of the POC can be loosely decomposed into four components to facilitate discussion:

1. Behaviors corresponding to host commands that preempt the regular behavioral flow.
2. Behavior that brings the POC to the *POC:ready* state.
3. Behavior leading from the *POC:ready* state to the *POC:normal active* state.
4. Behavior once the *POC:normal active* state has been reached, i.e., during "normal operation".

The remainder of this section addresses these four components in succession, explaining the required behavior using SDL diagrams.

2.3.3 Preempting commands

There are two commands (FREEZE and IMMEDIATE_READY) that are used to preempt the normal behavioral flow of the POC. They are depicted in Figure 2-6. It should be emphasized that these commands also apply to the behavior contained in the Wakeup and Startup macros (see Figure 2-12) that is detailed in Chapter 7.

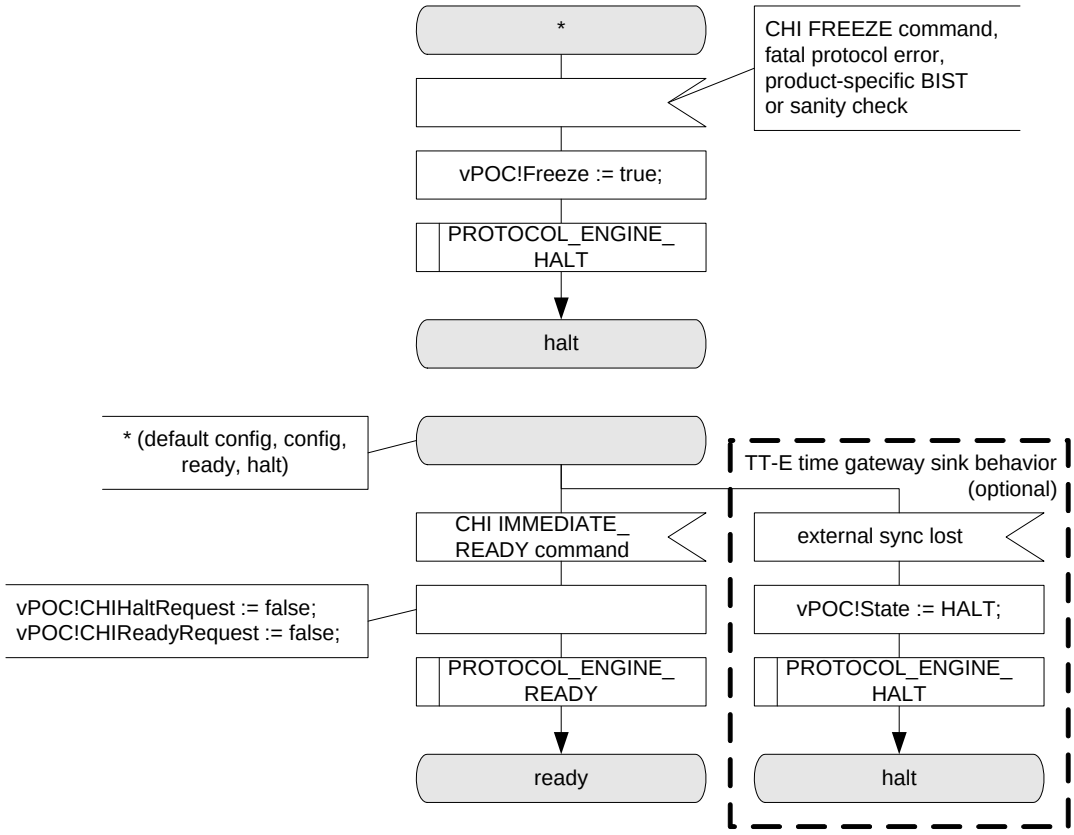

Figure 2-6: POC preempting immediate commands [POC].

When a serious error occurs, the POC is notified to halt the operation of the protocol engine. For this purpose, a freeze mechanism is supported. There are three methods for triggering the freeze mechanism:

1. A host FREEZE command relayed to the POC by the CHI.
2. A *fatal protocol error* signaled by the FSP process.
3. A product-specific error detected by a built-in self-test (BIST) or sanity check.

In all three circumstances the POC shall set *vPOC!Freeze* to true as an indicator that the event has occurred, stop the protocol engine by setting all core mechanism to the STANDBY mode, and then transition to the *POC:halt* state[17].

At the host's discretion, the ongoing operation of the POC can be interrupted by immediately placing the POC in the *POC:ready* state. In response to this command, the POC modes the core mechanisms appropriately (see section 2.3.1), and then transitions to *POC:ready*.

2.3.4 Deferred commands

2.3.4.1 DEFERRED_HALT, DEFERRED_READY and CLEAR_DEFERRED commands

The POC supports two deferred control commands that will postpone action until the end of a cycle or a time where the command can be processed with minimal disruption to ongoing processes.

[17] Note that the values of *vPOC!State* and *vPOC!StartupState* are intentionally not altered so that the CHI can indicate to the host what state the POC was in at the time the freeze occurred.

The CHI may relay the DEFERRED_HALT command from the host at any time the POC is in a state other than *POC:halt*. The CHI may relay the DEFERRED_READY command from the host at any time that it would be allowed to relay the IMMEDIATE_READY command (see Table 2-1 for details). If no communication is ongoing the effects of these commands is immediate. If communication is ongoing the effects of the commands are deferred, in most cases to the processing at the end of the cycle. If the POC is in the *POC:wakeup send* state a deferred command is processed after transmission of a complete WUP or detection of a wakeup collision. If the POC is in the *POC:coldstart collision resolution* state a deferred command is processed at the end of the cycle or after a frame header or a CAS is received. In all cases where the processing of the commands is deferred it is necessary to capture indications that the commands have occurred so that processing can take place at the appropriate time. Figure 2-7 depicts the procedure that captures these commands.

If an additional DEFERRED_HALT or DEFERRED_READY command is relayed from the CHI prior to the POC acting on a previous deferred command, the POC will only act upon the most recently received command.

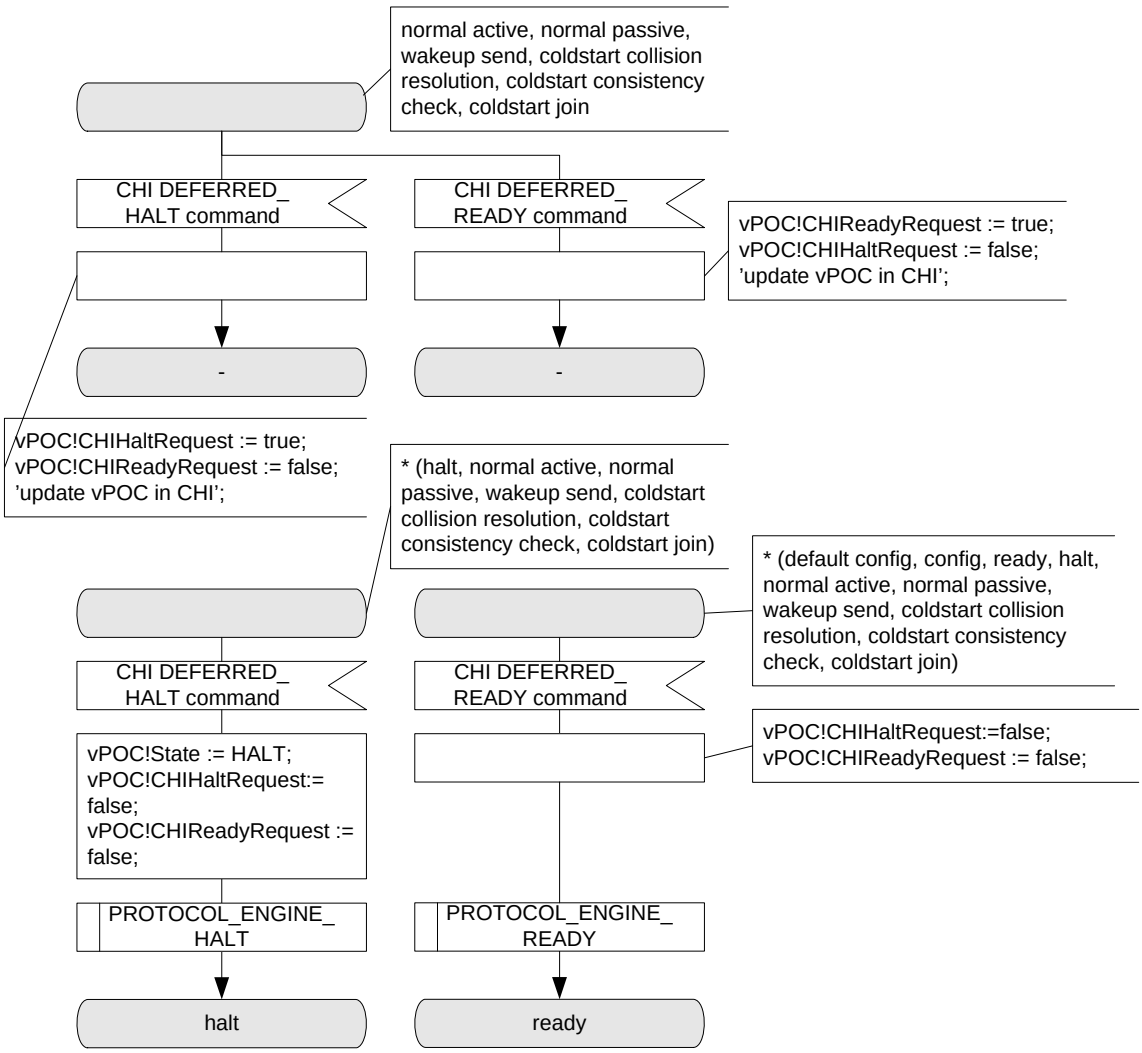

Figure 2-7: POC preempting deferred commands [POC].

The DEFERRED_HALT command shall be captured by setting the *vPOC!CHIHaltRequest* value to true if the command is not immediately processed. When processed at the end of the current cycle, the

DEFERRED_HALT command will cause the POC to enter the *POC:halt* state. This is the standard method used by the host to shut down the CC.

The DEFERRED_READY command shall be captured by setting the *vPOC!CHIReadyRequest* value to true if the command is not immediately processed. When processed at the end of the current cycle, the DEFERRED_READY command will cause the POC to enter the *POC:ready* state.

While in the *POC:wakeup send* state the DEFERRED_HALT command will cause the POC to enter the *POC:halt* state after transmission of a complete WUP or detection of a wakeup collision.

While in the *POC:wakeup send* state the DEFERRED_READY command will cause the POC to enter the *POC:ready* state after transmission of a complete WUP or detection of a wakeup collision.

Figure 2-8 depicts the HANDLE_DEFERRED_CHI_COMMANDS macro which is used in Chapter 7.

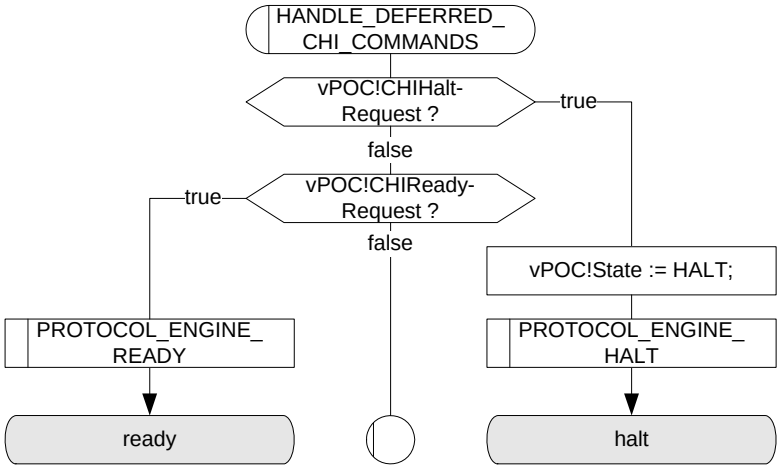

Figure 2-8: Macro to handle deferred CHI commands [POC].

Note that the macro shown in Figure 2-8 contains transitions to states that are defined outside of the macro (*POC:ready* and *POC:halt*). The reader should take care when interpreting this macro in higher level SDL diagrams, as the exits to other states will not appear in the higher-level diagram.

It is possible to delete a captured DEFERRED_HALT or DEFERRED_READY command as long as the POC has not yet reacted on the deferred command.

The CLEAR_DEFERRED command shall set the *vPOC!CHIReadyRequest* value and the *vPOC!CHIHalt-Request* value immediately to false. Note that the CLEAR_DEFERRED command is not able to delete a deferred command in all POC states because in a number of POC states a deferred command is executed immediately (see Table 2-1).

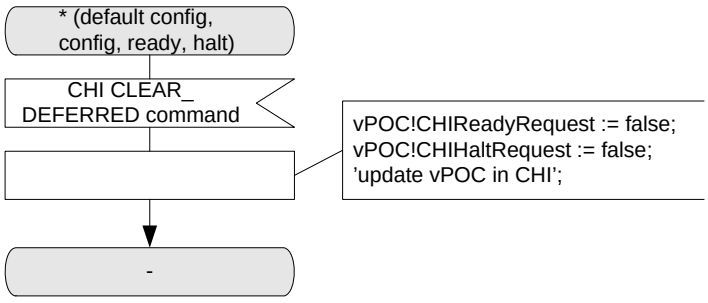

Figure 2-9: Cancelation of deferred commands [POC].

2.3.4.2 ALL_SLOTS command

The CHI may relay the ALL_SLOTS command from the host at any time while the POC is in the *POC:normal active* or *POC:normal passive* states. Its effect is realized during the processing at the end of the cycle, but it is necessary to capture an indication that the command has occurred so that appropriate processing will occur at cycle end. Figure 2-10 depicts the procedure that captures this command.

The ALL_SLOTS command shall be captured by setting *vPOC!SlotMode* to ALL_PENDING. The command shall be ignored if *vPOC!SlotMode* is not KEYSLOT. When processed at the end of the current cycle, the ALL_PENDING status causes the POC to enable the transmission of all frames for the node.

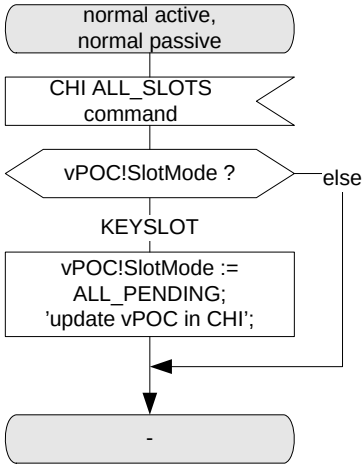

Figure 2-10: Capture of the ALL_SLOTS command for end-of-cycle processing [POC].

2.3.5 Reaching the *POC:ready* state

The tasks that the POC executes in order to reach the *POC:ready* state serve primarily as an initialization process for the POC and the core mechanisms. This initialization process is depicted in Figure 2-11.

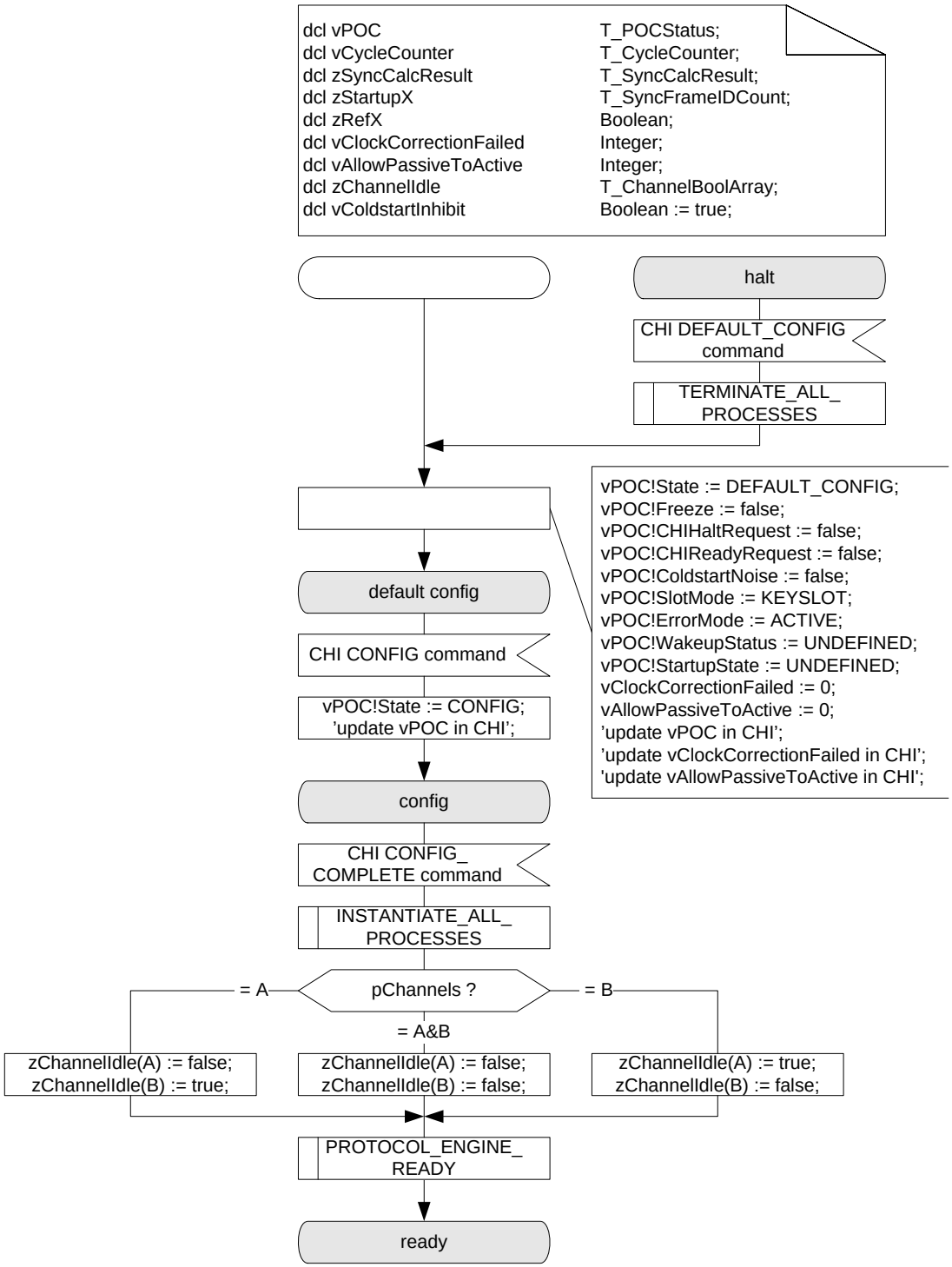

Figure 2-11: Reaching the *POC:ready* state [POC].

The POC shall enter the *POC:default config* state when the CC enters the *POC operational* state (see section 2.1.1). The *POC:default config* shall also be entered from the *POC:halt* state if a DEFAULT_CONFIG command is received from the CHI. In the latter case, the POC shall signal the core mechanisms to terminate so that they can be created again as a part of the normal initialization process.

Prior to entering the *POC:default config* state the POC shall initialize the elements of the *vPOC* data structure that are used to communicate the POC status to the CHI. With the exception of *vPOC!SlotMode*, the values assumed by the *vPOC* elements are obvious initial values and are depicted in Figure 2-11. The initial value of *vPOC!SlotMode* is defaulted to KEYSLOT until the configuration process is carried out to set it to the value desired by the host.

In the *POC:default config* state the POC awaits the explicit command from the host to enable configuration. The POC shall enter the *POC:config* state in response to the CONFIG command. Configuration of the CC is only allowed in the *POC:config* state and this state can only be entered with an explicit CONFIG command issued while the POC is in the *POC:default config* state or the *POC:ready* state (see section 2.3.6).

In the *POC:config* state the host configures the CC. The host is responsible for verifying this configuration and only allowing the initialization to proceed when a proper configuration is verified. For this purpose, an explicit CONFIG_COMPLETE command is required for the POC to progress from the *POC:config* state.

The POC shall transition to the *POC:ready* state in response to the CONFIG_COMPLETE command. On this transition, the POC shall create all of the core mechanism processes, incorporating the configuration values that were set in the *POC:config* state. It shall then update *vPOC* to reflect the new state, the newly configured value of slot mode[18], and the initial value of *vColdstartInhibit*. It shall then command all of the core mechanisms to their appropriate mode (see section 2.3.1). The POC then transitions to the *POC:ready* state.

2.3.5.1 Default configuration requirements

POC:default config is a state that ensures that the CC has a defined, stable default configuration prior to application-specific configuration that takes place in the *POC:config* state. Upon entry into the *POC:default config* state the CC must ensure that all configuration data or control data described in sections 9.3.1.1, 9.3.1.2.2, 9.3.2.5.1, and 9.3.2.10 are set to defined values as described below.

The default configuration that results from entry into the *POC:default config* state must have the characteristic that if a host makes no modification to the default configuration prior to the issuance of a RUN or WAKEUP command then the operation following the command will have no impact to ongoing communication on the cluster. The following configurations are required:

- All buffers (including FIFO buffers) shall be configured such that they can neither transmit nor receive.
- No slot shall be assigned for transmission or reception.
- The payload data valid flag of all message buffers shall be set to false.
- *pKeySlotID* and *pSecondKeySlotID* shall be set to 0.
- *pKeySlotUsedForStartup* shall be set to false.
- *pTwoKeySlotMode* shall be set to false.
- *pWakeupPattern* shall be set to 0.
- *pExternalSync* (if applicable) shall be set to false.
- No transmissions of WUDOP's or MTS's are scheduled (see section 9.3.1.2.2).

Other than the specific case described below, all other configuration data defined in sections 9.3.1.1, 9.3.2.5.1, and 9.3.2.10 shall be set to implementation dependent predefined initialization values. The initialization values for each individual configuration in the default configuration must be described in the documentation of the implementation.

[18] The value is determined by the node configuration, *pKeySlotOnlyEnabled*, a Boolean used to indicate whether the key slot only mode is enabled. This supports an optional strategy to limit frame transmissions following startup to the keyslots until the host confirms that the node is synchronized to the cluster and enables the remaining transmissions with an ALL_SLOTS command (see section 2.3.4.2).

An exception to the previous requirement is the configuration for *pChannels* (refer to section 9.3.1.1.2 for details). For this parameter there is no requirement for a transition into the *POC:default config* state to cause this configuration parameter to be set to any specific value (it is allowed, but not required).

2.3.6 Reaching the *POC:normal active* state

Following the initialization sequence (see section 2.3.5) the CC resides in the *POC:ready* state (see Figure 2-12). From this state the CC is able to perform the necessary tasks to start or join an actively communicating cluster. There are three POC actions that can take place, each of which is initiated by a specific command from the CHI. These commands are WAKEUP, RUN, and CONFIG.

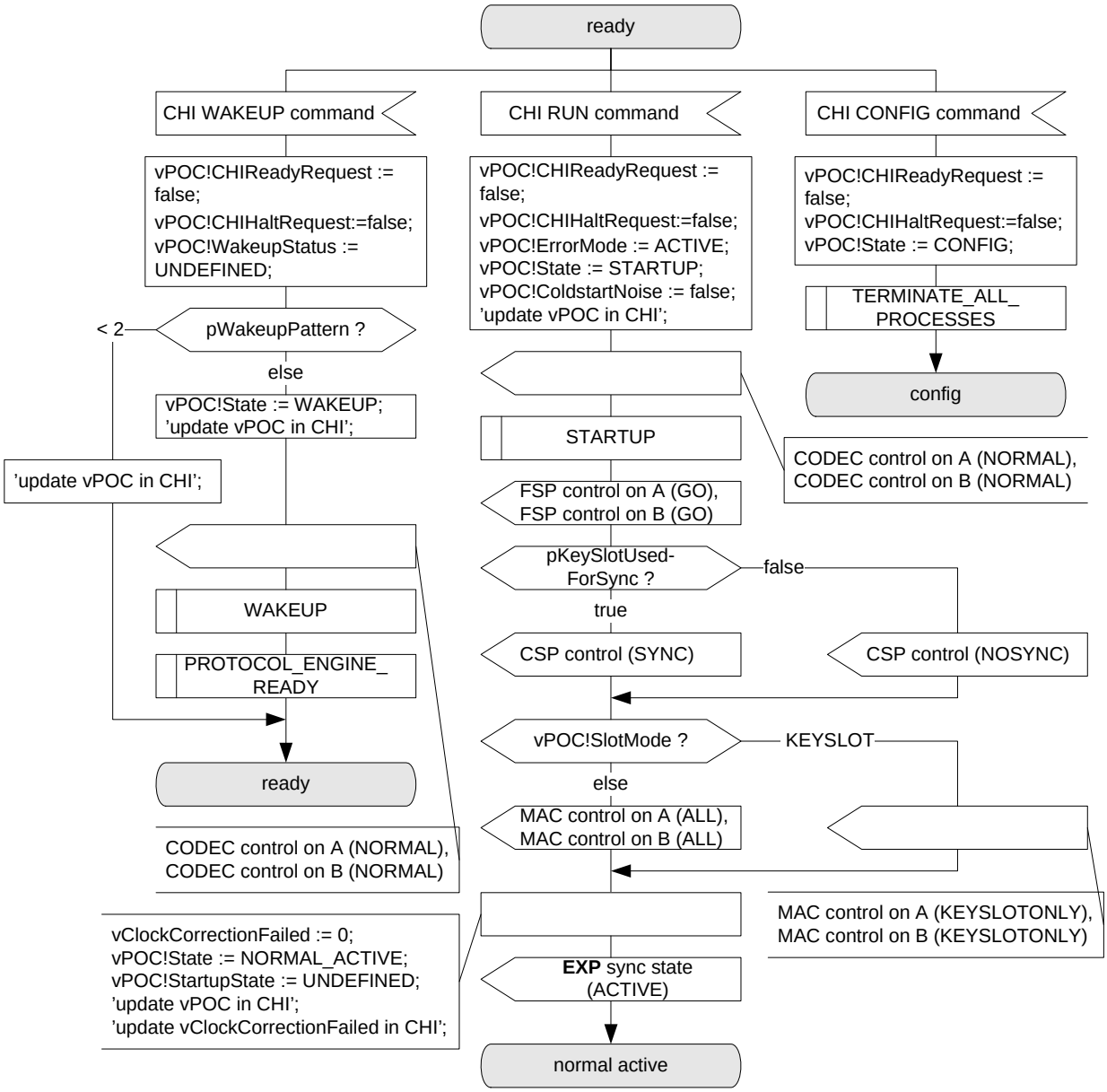

Figure 2-12: POC behavior in preparation for normal operation [POC].

The CONFIG command shall cause the host to re-enter the *POC:config* state to allow the host to alter the current CC configuration. Since the core mechanism processes are created on the transition back to

POC:ready following the configuration process, the processes shall be terminated on the transition to *POC:config*. This is accomplished in the SDL with the TERMINATE_ALL_PROCESSES macro (see section 2.3.1), which signals the individual processes so that they can terminate themselves.

The WAKEUP command shall cause the POC to commence the wakeup procedure in accordance with the configuration loaded into the CC when it was previously configured. This procedure is described in detail in section 7.1, and is represented in Figure 2-12 by the WAKEUP macro invocation. On completion of the wakeup procedure, the POC shall mode all the core mechanisms appropriately for *POC:ready* (see section 2.3.1) and return to the *POC:ready* state.

The RUN command shall cause the POC to commence a sequence of tasks to bring the POC to normal operation, i.e. the *POC:normal active* state. First, all internal status variables are reset to their starting values[19]. Then the startup procedure is executed. In Figure 2-12 this is represented by the STARTUP macro invocation. This procedure is described in detail in section 7.2. This procedure modes the core mechanisms appropriately to perform the sequence of tasks necessary for the node to start or enter an actively communicating cluster.

The startup procedure results in the node being synchronized to the timing of the cluster. At the end of the communication cycle, the POC shall mode the core mechanisms depending on the values of *vPOC!SlotMode* and the configuration *pKeySlotUsedForSync* as depicted in Figure 2-12:

1. The FSP mechanism shall be moded to GO for both channels.
2. If the node is a sync node (*pKeySlotUsedForSync* is true) CSP shall be moded to SYNC mode. Otherwise, CSP shall be moded to NOSYNC.
3. If the node is currently in key slot only mode (*vPOC!SlotMode* is KEYSLOT), then the POC shall mode the MAC to KEYSLOTONLY mode on both channels. If the node is not currently in key slot only mode (*vPOC!SlotMode* is ALL), then the POC shall mode the MAC to ALL mode on both channels.

The POC shall then enter the *POC:normal active* state.

2.3.6.1 Wakeup and startup support

As indicated above, the Wakeup and Startup procedures are performed in logical extensions of the POC that are embodied in the WAKEUP and STARTUP macros. The POC behavior captured in those macros is documented in Chapter 7 and is largely self-contained. However, there are two exceptions and they are depicted in Figure 2-13.

[19] This is necessary because the *POC:ready* state may have been entered due to a DEFERRED_READY or IMMEDIATE_READY command from the CHI that caused the POC to enter *POC:ready* from a state where the status variables had already been altered (see section 2.3.2).

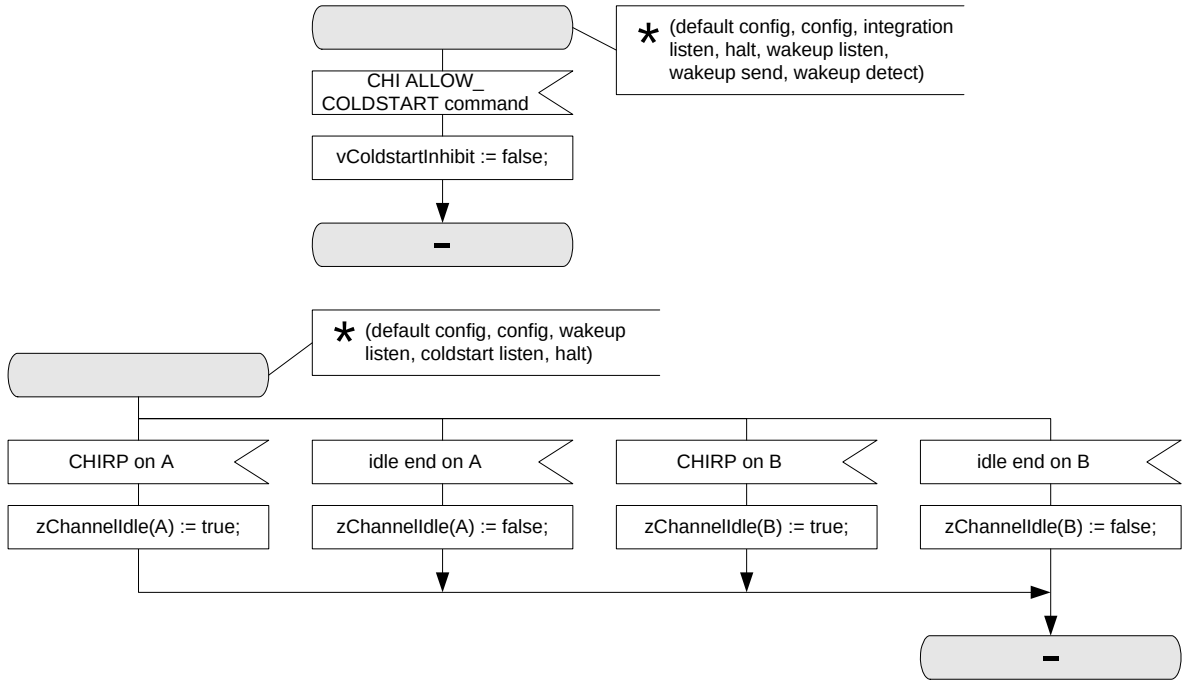

Figure 2-13: Conditions detected in support of the wakeup and startup procedures [POC].

The behavior of the POC during startup is influenced by whether the node is currently inhibited from acting as a leading coldstart node in the startup process (see section 7.2.3). A restriction on the node's ability to act as a leading coldstart node is reflected in the Boolean variable *vColdstartInhibit*. While this value is acted upon in the startup procedure, the CHI also allows the host to change the variable to false by issuing the ALLOW_COLDSTART command while in the *POC:ready* state and any of the states that are part of startup, normal active, or normal passive. Note that the POC sets the value of the *vColdstartInhibit* variable to true on all transitions into the *POC:ready* state. A system designer must be aware of this behavior, and must ensure that the ALLOW_COLDSTART commands are issued such that *vColdstartInhibit* has the desired value when the RUN command is issued.[20]

In a similar manner, both the wakeup and startup procedures must be able to determine whether or not a given channel is idle. Again, this knowledge is acted upon in the wakeup and startup procedures, but it can change at any point in time once the *POC:ready* state is reached. Hence it is relevant in the current context.

The channel idle status is captured using the mechanism depicted in Figure 2-13 and is stored in the appropriate element of the *zChannelIdle* array. The POC shall change the value of the appropriate *zChannelIdle* array element to false whenever a communication element start is signaled for the corresponding channel by the BITSTRB processes (see section 3.4.2). Similarly, the POC shall change the value of the appropriate *zChannelIdle* array element to true whenever a channel idle recognition point (CHIRP) is signaled for the corresponding channel by the BITSTRB processes (see section 3.4.2).

The *zChannelIdle* array is of type *T_ChannelBoolArray* as defined in Definition 2-7.

```
newtype T_ChannelBoolArray
      Array(T_Channel,Boolean);
```

[20] For example, if a WAKEUP command is issued after the host has already issued the ALLOW_COLDSTART command the *vColdstartInhibit* variable will be set to true at the completion of the wakeup attempt. A similar situation would occur if the host issues a CONFIG command after an ALLOW_COLDSTART command.

```
endnewtype;
```

Definition 2-7: Formal definition of T_ChannelBoolArray.

The index to the array is the channel identifier, which is of type *T_Channel* as defined in Definition 2-8.

```
newtype T_Channel
      literals A, B;
endnewtype;
```

Definition 2-8: Formal definition of T_Channel.

2.3.7 Behavior during normal operation

Other than the commands that preempt regular behavioral flow (see section 2.3.3), there are two components of the POC behavior once normal operation has begun.

- The capture of deferred host commands that the CHI relays to the POC for later processing (see section 2.3.4).
- The cyclical processing of error status information and deferred host commands at the end of each cycle.

2.3.7.1 Cyclical behavior

When the *POC:normal active* state is reached, the protocol's core mechanisms are set to the modes appropriate for performing the communication tasks for which the CC is intended. In the absence of atypical influences, the POC will remain in the *POC:normal active* state until the host initiates the shutdown process by issuing a command that causes a transition to the *POC:halt* or *POC:ready* state either immediately or at the cycle boundary.

While in the *POC:normal active* state, the POC performs several tasks at the end of each communication cycle to determine if it is necessary to change its own operating mode or the operating modes of any of the core mechanisms. These changes result in appropriate moding commands to the core mechanisms. The remainder of this section describes the cyclical POC processing that evaluates whether there is a need for these mode changes and the moding consequences.

2.3.7.1.1 Cycle counter

The moding decisions made by the POC at the end of each cycle depend on whether the current cycle number is even or odd. At the start of each cycle, the clock synchronization mechanism signals the current cycle number to the POC with the *cycle start* signal so that the POC can make this determination. The POC shall acquire the current cycle count as depicted in Figure 2-14.

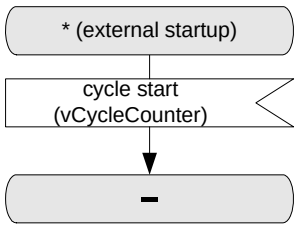

Figure 2-14: POC determination of the cycle counter value [POC].

2.3.7.1.2 *POC:normal active* state

Following a successful startup the POC will reside in the *POC:normal active* state (see section 2.3.6). As depicted in Figure 2-15 the POC performs a sequence of tasks at the end of each communication cycle for the purpose of determining whether the POC should change the moding of the core mechanisms before the

beginning of the next communication cycle. The CSP process (see Figure 8-6) signals the completion of the clock correction calculation to the POC using the *SyncCalcResult* signal. This signal results in the following:

1. If *vPOC!SlotMode* is ALL_PENDING, the POC shall change its value to ALL and enable all frame transmissions by moding MAC to ALL for both channels. This completes the POC's reaction to the ALL_SLOTS command received asynchronously during the preceding cycle (see section 2.3.4.2).
2. The POC then performs a sequence of error checking tasks whose outcome determines the subsequent behavior. This task sequence is represented by the invocation of the NORMAL_ERROR_CHECK macro in Figure 2-15. The details of this task sequence are described in section 2.3.7.1.4.2. As a result the POC will be in one of the following states:
   a. If the *vPOC!ErrorMode* is ACTIVE and the CHI did not relay a DEFERRED_HALT or a DEFERRED_READY command to the POC in the preceding communication cycle (see section 2.3.4.1), the POC shall remain in the *POC:normal active* state.
   b. If the *vPOC!ErrorMode* is PASSIVE and the CHI did not relay a DEFERRED_HALT or a DEFERRED_READY command to the POC in the preceding communication cycle (see section 2.3.4.1), the POC shall mode the MAC and CSP to halt frame transmission and transition to the *POC:normal passive* state.
   c. If *vPOC!ErrorMode* is COMM_HALT the POC shall halt the execution of the core mechanisms by moding them to STANDBY and transition to the *POC:halt* state.
   d. If the CHI did relay a DEFERRED_HALT command to the POC in the preceding communication cycle (see section 2.3.4.1), the POC shall stop the execution of the core mechanisms by moding them to STANDBY and transition to the *POC:halt* state.
   e. If the CHI did relay a DEFERRED_READY command to the POC in the preceding communication cycle (see section 2.3.4.1), the POC shall stop the execution of the core mechanisms by moding them to STANDBY and transition to the *POC:ready* state.

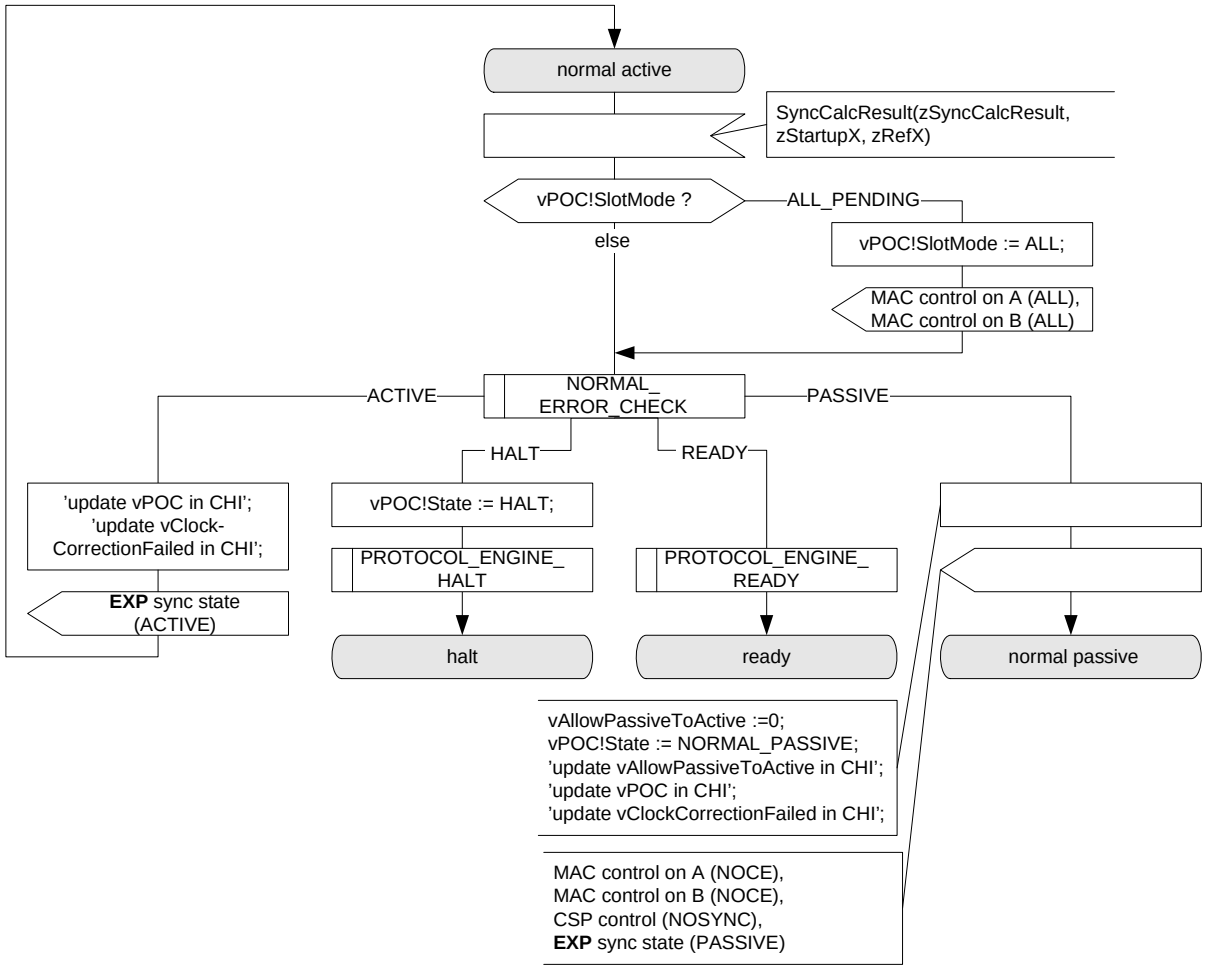

Figure 2-15: Cyclical behavior in the *POC:normal active* state [POC].[21]

2.3.7.1.3 *POC:normal passive* state

The POC's behavior in the *POC:normal passive* state is analogous its behavior in the *POC:normal active* state (see section 2.3.7.1.2). As depicted in Figure 2-16 the POC performs a sequence of tasks at the end of each communication cycle for the purpose of determining whether the POC should change the moding of the core mechanisms before the beginning of the next communication cycle. The CSP process (see Figure 8-6) signals the completion of the clock correction calculation to the POC using the *SyncCalcResult* signal. This signal results in the following:

1. If *vPOC!SlotMode* is ALL_PENDING, the POC shall change its value to ALL and enable all frame transmissions by moding Media Access Control process to ALL for both channels. This completes the POC's reaction to the ALL_SLOTS command received asynchronously during the preceding cycle (see section 2.3.4.2).
2. The POC then performs a sequence of error checking tasks whose outcome determines the subsequent behavior. This task sequence is represented by the invocation of the PASSIVE_ERROR_CHECK macro in Figure 2-16. The details of this task sequence are described in section 2.3.7.1.4.3. As a result the POC will be in one of the following states:

[21] *zStartupX* is *zStartupNodes* in even cycles and *zRxStartupPairs* in odd cycles. *zRefX* is *zRefNode* in even cycles and *zRefPair* in odd cycles. See Figure 8-6 for details.

a. If the *vPOC!ErrorMode* is ACTIVE and the CHI did not relay a DEFERRED_HALT or a DEFERRED_READY command to the POC in the preceding communication cycle (see section 2.3.4.1), the POC shall mode the MAC and CSP mechanisms to support resumption of frame transmission based on whether the node is a sync node (*pKeySlotUsedForSync* is true) and whether the node is currently in key slot only mode, and then transition to the *POC:normal active* state.

b. If the *vPOC!ErrorMode* is PASSIVE and the CHI did not relay a DEFERRED_HALT or a DEFERRED_READY command to the POC in the preceding communication cycle (see section 2.3.4.1), the POC shall remain in to the *POC:normal passive* state.

c. If *vPOC!ErrorMode* is COMM_HALT the POC shall stop the execution of the core mechanisms by moding them to STANDBY and transition to the *POC:halt* state.

d. If the CHI did relay a DEFERRED_HALT command to the POC in the preceding communication cycle (see section 2.3.4.1), the POC shall stop the execution of the core mechanisms by moding them to STANDBY and transition to the *POC:halt* state.

e. If the CHI did relay a DEFERRED_READY command to the POC in the preceding communication cycle (see section 2.3.4.1), the POC shall stop the execution of the core mechanisms by moding them to STANDBY and transition to the *POC:ready* state.

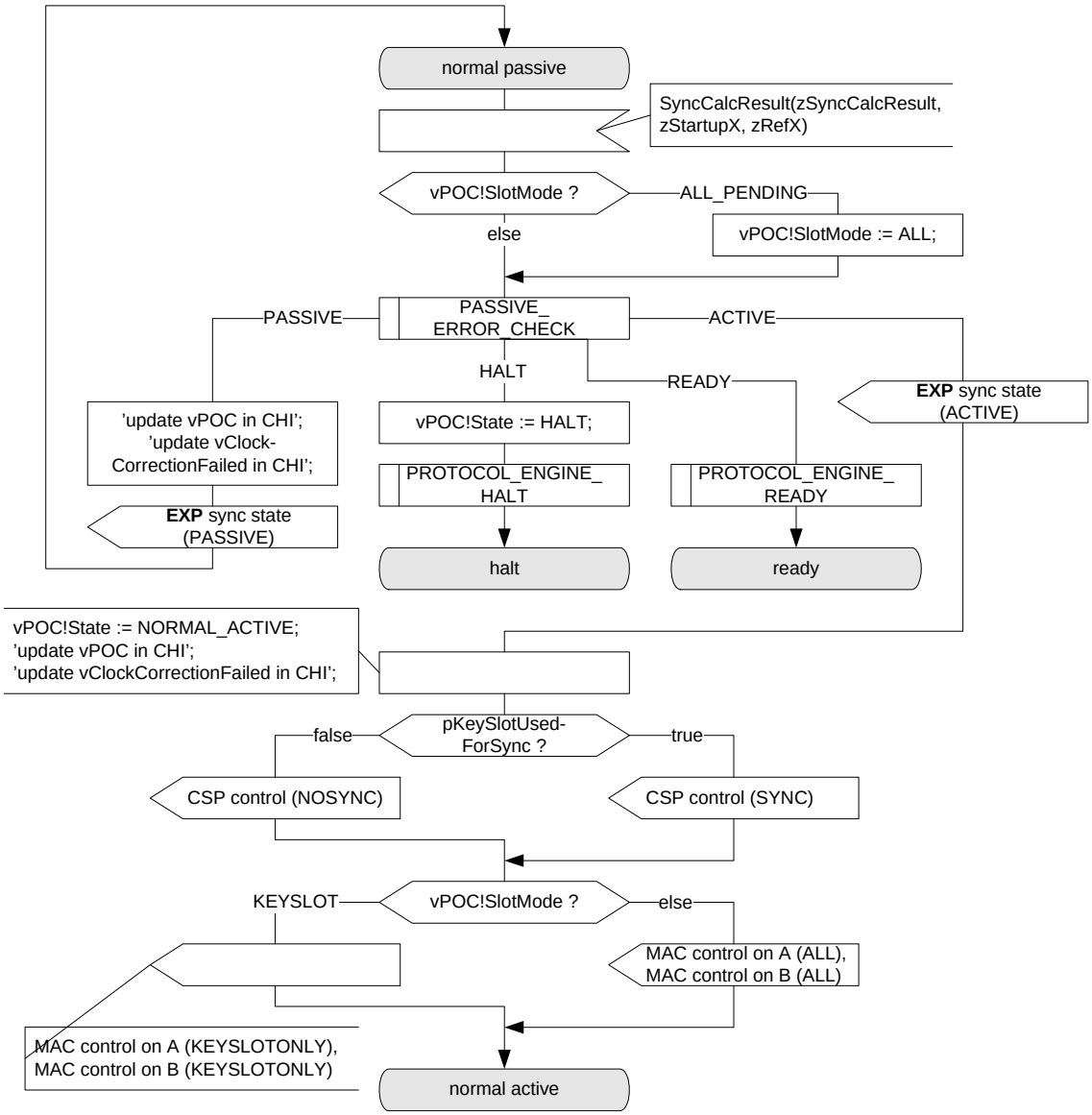

Figure 2-16: Cyclical behavior in the *POC:normal passive* state [POC].[22]

2.3.7.1.4 Error checking during normal operation

During normal operation (POC is in the *POC:normal active* or *POC:normal passive* state) error checking is performed by two similarly structured procedures described by the NORMAL_ERROR_CHECK (see Figure 2-17) and PASSIVE_ERROR_CHECK (see Figure 2-18) macros. In both cases, the macro determines a new value of *vPOC!ErrorMode* which, in turn, determines the subsequent cycle-end behavior (see sections 2.3.7.1.2 and 2.3.7.1.3).

[22] *zStartupX* is *zStartupNodes* in even cycles and *zRxStartupPairs* in odd cycles. *zRefX* is *zRefNode* in even cycles and *zRefPair* in odd cycles. See Figure 8-6 for details.

2.3.7.1.4.1 Error checking overview

At the end of each communication cycle CSP communicates the error consequences of the clock synchronization mechanism's rate and offset calculations. In the SDL this is accomplished with the *SyncCalcResult* signal whose first argument, *zSyncCalcResult*, assumes one of three values:

1. WITHIN_BOUNDS indicates that the calculations resulted in no errors.
2. MISSING_TERM indicates that either the rate or offset correction could not be calculated.
3. EXCEEDS_BOUNDS indicates that either the rate or offset correction term calculated was deemed too large when compared to the calibrated limits.

The consequences of the EXCEEDS_BOUNDS value are processed in every cycle. The other two results are only processed at the end of odd cycles.

The error checking behavior is detailed in sections 2.3.7.1.4.2 and 2.3.7.1.4.3. A number of configuration alternatives and the need to verify cycle timing before resuming communication influence the detailed error checking behavior. However, the basic concept can be grasped by considering the behavior in the absence of these considerations. In the absence of these influences, the processing path is determined by the value of *zSyncCalcResult* as follows:

1. In all cycles (even or odd), *zSyncCalcResult* = EXCEEDS_BOUNDS causes the POC to transition to the *POC:halt* state.
2. In odd cycles, *zSyncCalcResult* = WITHIN_BOUNDS causes the POC to stay in, or transition to, the *POC:normal active* state.
3. In odd cycles, if *zSyncCalcResult* = MISSING_TERM
   a. The POC will transition to the *POC:halt* state if the MISSING_TERM value has persisted for at least *gMaxWithoutClockCorrectionFatal* odd cycles.
   b. The POC will transition to (or remain in) the *POC:normal passive* state if the MISSING_TERM value has persisted for at least *gMaxWithoutClockCorrectionPassive*, but less than *gMaxWithoutClockCorrectionFatal* odd cycles.
   c. Otherwise the POC stays in the *POC:normal active* state.

2.3.7.1.4.2 Error checking details for the *POC:normal active* state

The *zSyncCalcResult* value obtained from CSP is used to determine the new *vPOC!ErrorMode* as depicted in Figure 2-17.

1. If *zSyncCalcResult* is WITHIN_BOUNDS, the *vPOC!ErrorMode* remains ACTIVE.
2. If *zSyncCalcResult* is EXCEEDS_BOUNDS:
   a. If the node is configured to allow communication to be halted due to severe clock calculation errors (*pAllowHaltDueToClock* is true), then the *vPOC!ErrorMode* is set to COMM_HALT.
   b. If the node is configured not to allow communication to be halted due to severe clock calculation errors (*pAllowHaltDueToClock* is false), then the *vPOC!ErrorMode* is set to PASSIVE.
3. If *zSyncCalcResult* is MISSING_TERM and the cycle is even, the condition is ignored and *vPOC!ErrorMode* remains ACTIVE.
4. If *zSyncCalcResult* is MISSING_TERM and the cycle is odd the behavior is determined by how many consecutive odd cycles (*vClockCorrectionFailed*) have yielded MISSING_TERM.
   a. If *vClockCorrectionFailed* < *gMaxWithoutClockCorrectionPassive*, then the *vPOC!ErrorMode* remains ACTIVE.
   b. If *gMaxWithoutClockCorrectionPassive* <= *vClockCorrectionFailed* < *gMaxWithoutClockCorrectionFatal*, then the *vPOC!ErrorMode* is set to PASSIVE.
   c. If *vClockCorrectionFailed* >= *gMaxWithoutClockCorrectionFatal* and

i. The node is configured to allow communication to be halted due to severe clock calculation errors (*pAllowHaltDueToClock* is true), then the *vPOC!ErrorMode* is set to COMM_HALT.

ii. The node is configured not to allow communication to be halted due to severe clock calculation errors (*pAllowHaltDueToClock* is false), then the *vPOC!ErrorMode* is set to PASSIVE.

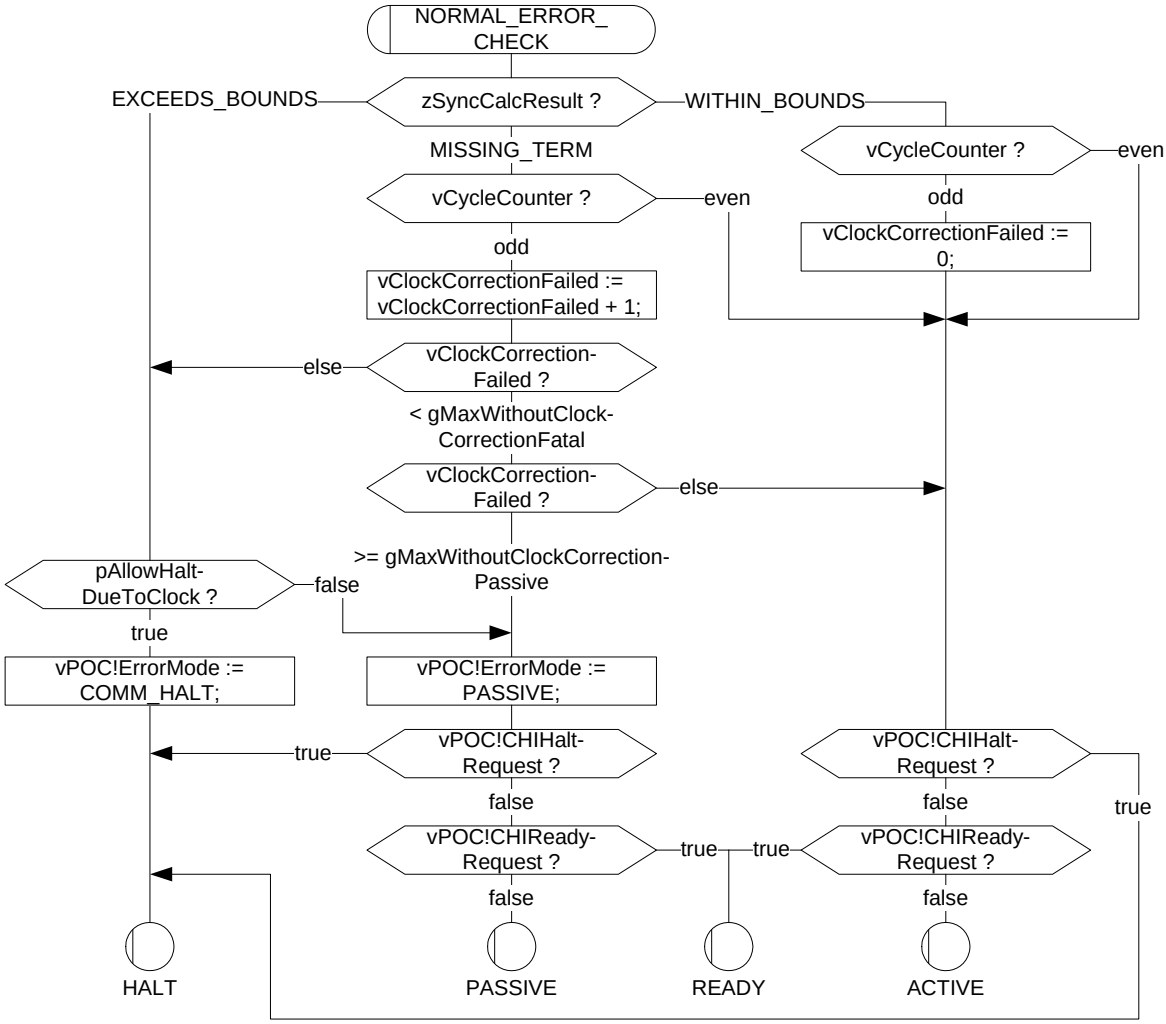

Figure 2-17: Error checking in the *POC:normal active* state [POC].

2.3.7.1.4.3 Error checking details for the *POC:normal passive* state

The *zSyncCalcResult* value obtained from CSP is used to determine the new *vPOC!ErrorMode* as depicted in Figure 2-18.

1. If *zSyncCalcResult* is WITHIN_BOUNDS and it is an even cycle the condition is ignored and *vPOC!ErrorMode* remains PASSIVE.
2. If *zSyncCalcResult* is WITHIN_BOUNDS and it is an odd cycle:
   a. If the node is configured to disallow the resumption of transmissions following the entry to *POC:normal passive* (*pAllowPassiveToActive* is zero) *vPOC!ErrorMode* remains PASSIVE.

b. If the node is configured to allow the resumption of transmissions following the entry to *POC:normal passive* (*pAllowPassiveToActive* is non-zero) the behavior is determined by how many consecutive odd cycles have yielded WITHIN_BOUNDS.

i. If less than *pAllowPassiveToActive* consecutive odd cycles have yielded WITHIN_BOUNDS, then the *vPOC!ErrorMode* remains PASSIVE.

ii. If at least *pAllowPassiveToActive* consecutive odd cycles have yielded WITHIN_BOUNDS, the *vPOC!ErrorMode* depends on the number (*zStartupX*) of startup frame pairs observed in the preceding double cycle.

A. If the node has seen more than one startup frame pair (*zStartupX* > 1) then the *vPOC!ErrorMode* is set to ACTIVE.

B. If the node has seen only one startup frame pair (*zStartupX* = 1) and if the node is a coldstart node (*pKeySlotUsedForStartup* = true) then the *vPOC!ErrorMode* is set to ACTIVE.

C. If neither of the preceding two conditions is met then the *vPOC!ErrorMode* remains PASSIVE.

3. If *zSyncCalcResult* is EXCEEDS_BOUNDS:

a. If the node is configured to allow communication to be halted due to severe clock calculation errors (*pAllowHaltDueToClock* is true), then the *vPOC!ErrorMode* is set to COMM_HALT.

b. If the node is configured not to allow communication to be halted due to severe clock calculation errors (*pAllowHaltDueToClock* is false), then the *vPOC!ErrorMode* remains PASSIVE.

4. If *zSyncCalcResult* is MISSING_TERM and the cycle is even, the condition is ignored and *vPOC!ErrorMode* remains PASSIVE.

5. If *zSyncCalcResult* is MISSING_TERM and the cycle is odd, then the behavior is determined by how many consecutive odd cycles have yielded MISSING_TERM.

a. If at least *gMaxWithoutClockCorrectionFatal* consecutive odd cycles have yielded MISSING_TERM, and

i. The node is configured to allow communication to be halted due to severe clock calculation errors (*pAllowHaltDueToClock* is true), then the *vPOC!ErrorMode* is set to COMM_HALT.

ii. The node is configured not to allow communication to be halted due to severe clock calculation errors (*pAllowHaltDueToClock* is false), then the *vPOC!ErrorMode* remains PASSIVE.

b. If less than *gMaxWithoutClockCorrectionFatal* consecutive odd cycles have yielded MISSING_TERM then the *vPOC!ErrorMode* remains PASSIVE.

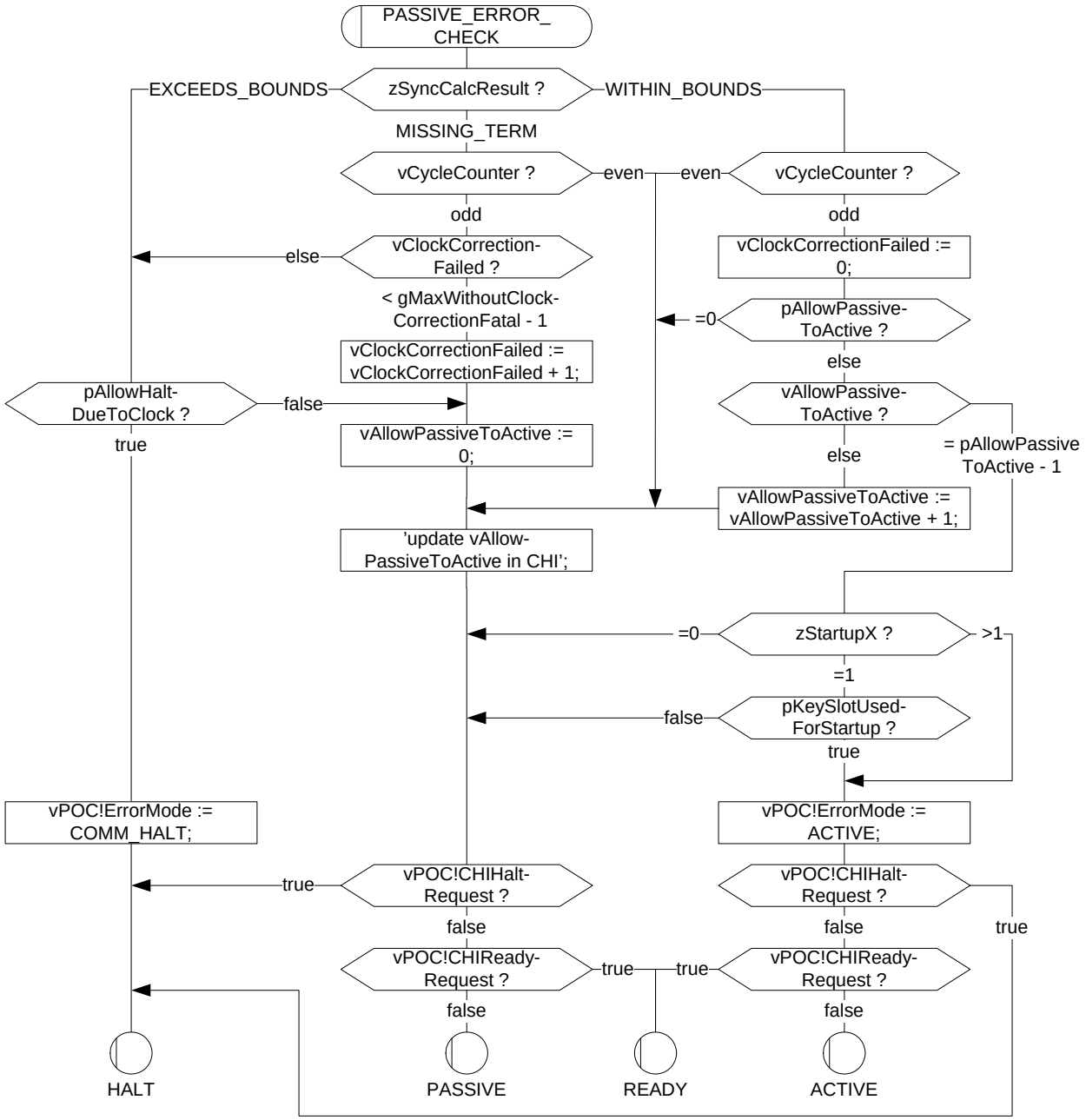

Figure 2-18: Error checking in the *POC:normal passive* state [POC].[23]

[23] *zStartupX* is *zStartupNodes* in even cycles and *zRxStartupPairs* in odd cycles. *zRefX* is *zRefNode* in even cycles and *zRefPair* in odd cycles. See Figure 8-6 for details.

Chapter 3
Coding and Decoding

This chapter defines how the node performs frame and symbol coding and decoding.

3.1 Principles

This chapter describes the coding and decoding behavior of the TxD, RxD, and TxEN interface signals between the communication controller and the bus driver.

A node may support up to two independent physical layer channels, identified as channel A and channel B. Refer to section 1.12.4 for additional information on the interface between the CC and the BD. The description in this chapter assumes a physical layer that appears to the protocol engine as a binary medium with two distinct levels, called HIGH and LOW.[24] A bit stream generated from these two levels is called a communication element (CE).

A node shall use a non-return to zero (NRZ) signaling method for coding and decoding of a CE. This means that the generated bit level is either LOW or HIGH during the entire bit time *gdBit*.

The node processes bit streams present on the physical media, extracts frame and symbol information, and passes this information to the relevant FlexRay processes.

3.2 Description

In order to support two channels each node must implement two sets of independent coding and decoding processes, one for channel A and another for channel B. The subsequent paragraphs of this chapter specify the function of the coding and decoding for channel A. It is assumed that whenever a channel-specific process is defined for channel A there is another, essentially identical, process defined for channel B, even though this process is not explicitly described in the specification.The description of the coding and decoding behavior is contained in three processes. These processes are the main coding and decoding process (CODEC) and the following two sub-processes:

1. Bit strobing process (BITSTRB).
2. Wakeup pattern decoding process (WUPDEC).

The POC is responsible for creating the CODEC process before entering the *POC:ready* state. Once instantiated, the CODEC process is responsible for creating and terminating the subprocesses. The POC is responsible for sending a signal that causes a termination of the CODEC process. If the CODEC process is not executing (i.e., if it has not yet been instantiated or if it has been terminated) the TxEN and TxD outputs shall be HIGH.[25]

The relationships between the coding/decoding and the other core mechanisms are depicted in Figure 3-1[26].

[24] Detailed bus state definitions may be found in the appropriate physical layer specification ([EPL10], for example).

[25] When a dual channel device is configured to operate in a single channel mode, the internal TxEN and TxD signals must be driven high on the unused channel. This requirement applies whenever the internal signals TxEN and TxD are connected to the external signals TxEN_external and TxD_external (see section 1.12.4).

[26] The dark lines represent data flows between mechanisms that are relevant to this chapter. The lighter gray lines are relevant to the protocol, but not to this chapter.

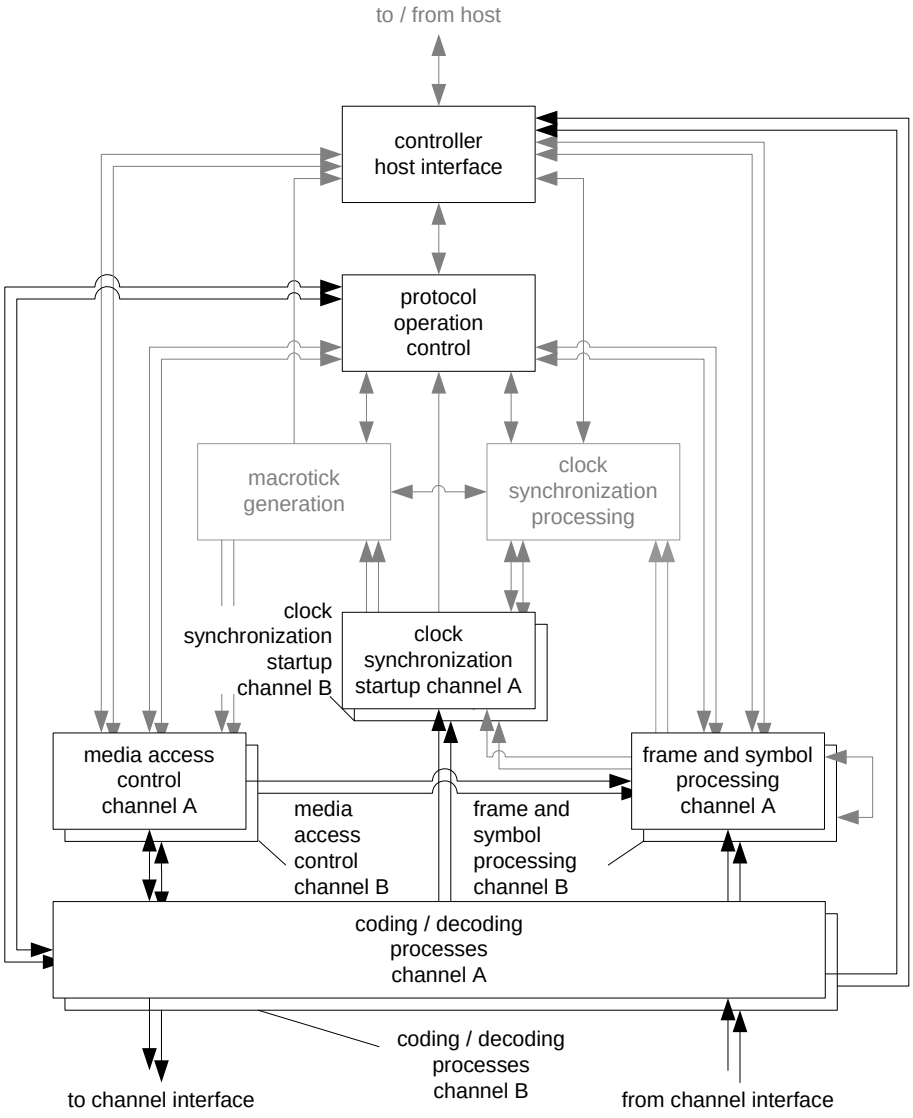

Figure 3-1: Coding / Decoding context.

3.2.1 Frame and symbol encoding

This section specifies the behavior of the mechanisms used by the node to encode the communication elements into a bit stream and how the transmitting node represents this bit stream to the bus driver for communication onto the physical media.

3.2.1.1 Frame encoding

3.2.1.1.1 Transmission start sequence

The transmission start sequence (TSS) is used to initiate proper connection setup through the network. A transmitting node generates a TSS that consists of a continuous LOW for a period given by the parameter *gdTSSTransmitter*.

The purpose of the TSS is to mark the beginning of a transmission and to set up the path between the transmitter and receiver. This includes setting up the input and output connections of an active star as well as allowing the receiving bus driver the time necessary to realize that the bus is no longer idle. Further, the active low portion of the TSS allows a transmitting BD to actually begin transmission (since the BD will not actually begin transmitting after TxEN is activated until TxD is commanding an active low). During this set up, active stars and bus drivers truncate a number of bits at the beginning of a communication element. The TSS prevents the content of the frame or symbol[27] from being truncated.

3.2.1.1.2 Frame start sequence

The frame start sequence (FSS) is used to compensate for a possible quantization error in the first byte start sequence after the TSS. The FSS shall consist of one HIGH *gdBit* time. The node shall append an FSS to the bit stream immediately following the TSS of a transmitted frame.

3.2.1.1.3 Byte start sequence

The byte start sequence (BSS) is used to provide bit stream timing information to the receiving devices. The BSS shall consist of one HIGH *gdBit* time followed by one LOW *gdBit* time. Each byte of frame data shall be sent on the channel as an extended byte sequence that consists of one BSS followed by eight data bits.

3.2.1.1.4 Frame end sequence

The frame end sequence (FES) is used to mark the end of the last byte sequence of a frame. The FES shall consist of one LOW *gdBit* time followed by one HIGH *gdBit* time. The node shall append an FES to the bit stream immediately after the last extended byte sequence of the frame.

For frames transmitted in the static segment the second bit of the FES is the last bit in the transmitted bit stream. As a result, the transmitting node shall set the TxEN signal to HIGH at the end of the second bit of the FES.

For frames transmitted in the dynamic segment the FES is followed by the dynamic trailing sequence (see below).

3.2.1.1.5 Dynamic trailing sequence

The dynamic trailing sequence (DTS), which is only used for frames transmitted in the dynamic segment, is used to indicate the exact point in time of the transmitter's minislot action point[28] and prevents premature channel idle detection[29] by the receivers. When transmitting a frame in the dynamic segment the node shall transmit a DTS immediately after the FES of the frame.

The DTS consists of two parts - a variable-length period with the TxD output at the LOW level, followed by a fixed-length period with the TxD output at the HIGH level. The minimum length of the LOW period is one *gdBit*. After this minimum length the node leaves the TxD output at the LOW level until the next minislot action point. At the next minislot action point the node shall switch the TxD output to the HIGH level, and the node shall switch the TxEN output to the HIGH level after a delay of 1 *gdBit* after the minislot action point[30]. The duration of a DTS is variable and can assume any value between 2 *gdBit* (for a DTS composed of 1 *gdBit* LOW and 1 *gdBit* HIGH), and *gdMinislot* + 2 *gdBit* (if the DTS starts slightly later than one *gdBit* before a minislot action point).

Note that the processes defining the behavior of the CODEC do not have any direct knowledge of the minislot action point - those processes are informed of the appropriate time to end the generation of the DTS by the signal *stop transmission on A* sent by the MAC process.

---

[27] Note that a TSS is not used before transmission of a WUS or WUDOP because those symbols begin with a sufficiently long active low phase to achieve these goals.

[28] See also Chapter 5.

[29] See also section 3.2.5.

[30] This ensures that there is a period of one *gdBit* during which the TxD output is at the HIGH level prior to the transition of TxEN output to the HIGH level. This is required for the stability of certain types of physical layers.

3.2.1.1.6 Frame bit stream assembly

In order to transmit a frame the node assembles a bit stream out of the frame data using the elements described above. The behavior, which is described by the CODEC process (see Figure 3-19) consists of the following steps:

1. Break the frame data down into individual bytes.
2. Prepend a TSS at the start of the bit stream.
3. Add an FSS at the end of the TSS.
4. Create extended byte sequences for each frame data byte by adding a BSS before the bits of the byte.
5. Assemble a continuous bit stream for the frame data by concatenating the extended byte sequences in the same order as the frame data bytes.
6. Calculate the bytes of the frame CRC, create extended byte sequences for these bytes, and concatenate them to form a bit stream for the frame CRC.
7. Append an FES at the end of the bit stream.
8. Append a DTS after the FES (if the frame is to be transmitted in the dynamic segment).

Steps 1 - 6 of the list above are performed by the **prepbitstream** function used by the CODEC process (see Figure 3-19).

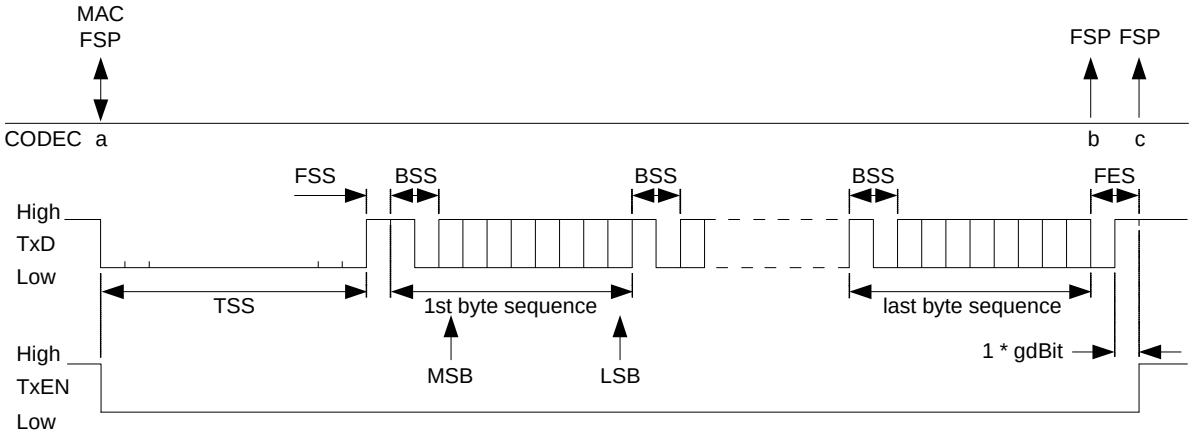

Figure 3-2: Frame encoding in the static segment.

Figure 3-2 shows the bit stream of a frame transmitted in the static segment and related events relevant to the CODEC process:

a. Input signal *transmit frame on A (vCEType, vTF)* received from the MAC process (see Figure 5-16) and output signal *decoding halted on A* sent to the FSP process (see Figure 6-10, Figure 6-11, and Figure 6-18).

b. Output signal *frame transmitted on A* sent to the FSP process (see Figure 6-19).

c. Output signal *decoding started on A* sent to the FSP process (see Figure 6-19).

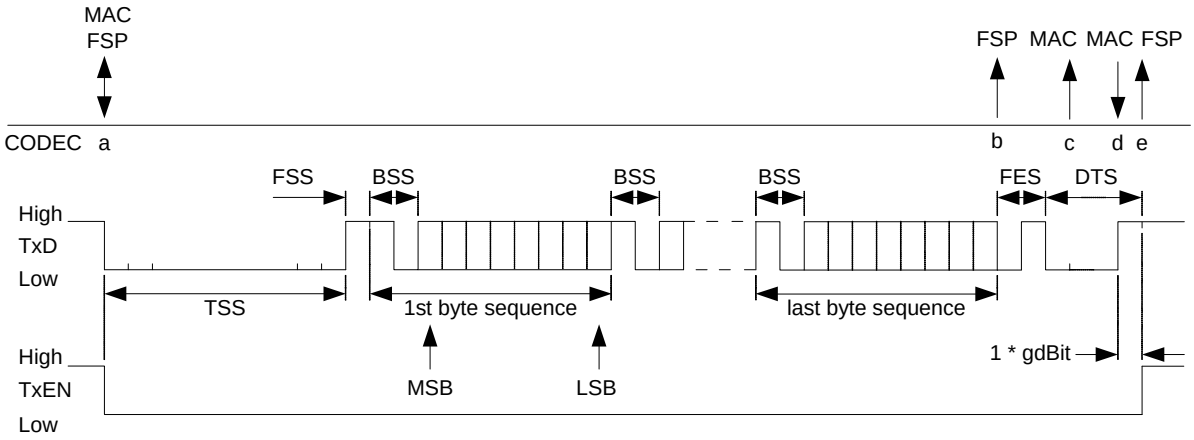

Figure 3-3: Frame encoding in the dynamic segment.

Figure 3-3 shows the bit stream of a frame transmitted in the dynamic segment and related events relevant to the CODEC process:

a. Input signal *transmit frame on A (vCEType, vTF)* received from the MAC process (see Figure 5-22) and output signal *decoding halted on A* sent to the FSP process (see Figure 6-10, Figure 6-11, and Figure 6-18).
b. Output signal *frame transmitted on A* sent to the FSP process (see Figure 6-19).
c. Output signal *DTS start on A* sent to the MAC process (see Figure 5-22).
d. Input signal *stop transmission on A* received from the MAC process (see Figure 5-22).
e. Output signal *decoding started on A* sent to the FSP process (see Figure 6-19).

3.2.1.2 Symbol encoding

The FlexRay communication protocol defines four symbols that are represented by three distinct symbol bit patterns:

1. Pattern 1 = Collision Avoidance Symbol[31] (CAS) and Media Access Test Symbol (MTS).
2. Pattern 2 = Wakeup Symbol (WUS).
3. Pattern 3 = Wakeup During Operation Pattern (WUDOP).

The node shall encode the MTS and CAS in exactly the same manner. Receivers distinguish between these symbols based on the node's protocol status. The encoding process does not distinguish between these two symbols. The bit streams for each of the symbols are described in the subsequent sections.

3.2.1.2.1 Collision avoidance symbol and media access test symbol

The node shall transmit these symbols starting with the TSS, followed by a LOW level with a duration of *cdCAS* as shown in Figure 3-4. The node shall transmit these symbols with the edges of the TxEN signal being synchronous with the TxD at the start of transmission, and with TxD returning to high after a delay period (defined by the parameter *cdStaggerDelay*) following the point that the TxEN signal returns to high at the end of transmission. For details refer to Figure 3-18 and Figure 3-24.[32]

[31] See also Chapter 7.

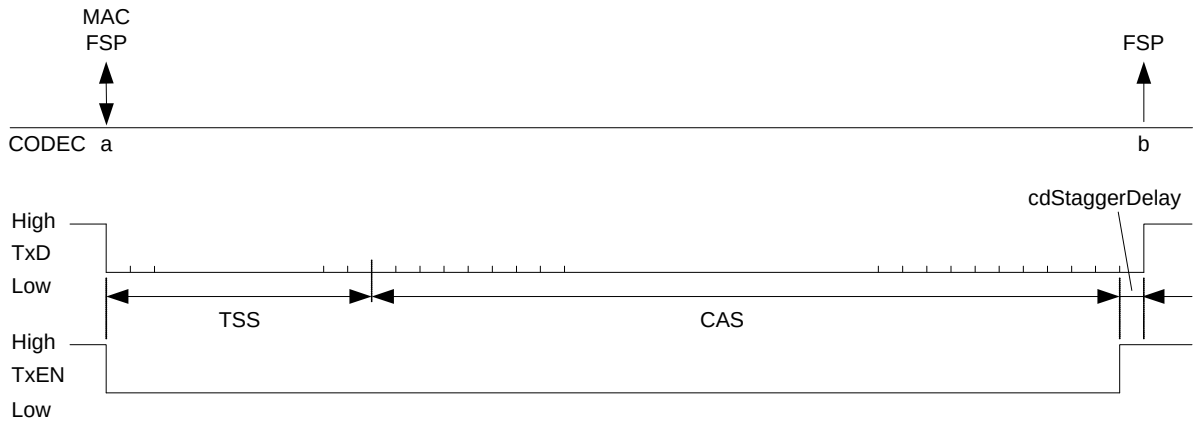

Figure 3-4: CAS and MTS symbol encoding.

Figure 3-4 illustrates the bit stream for a CAS or MTS symbol and related events relevant to the CODEC process:

a. Input signal *transmit symbol on A (vCEType)* received from the MAC process with *vCEType* = CAS_MTS (from Figure 5-13 if the node is sending a CAS or from Figure 5-26 if the node is sending an MTS) and output signal *decoding halted on A* sent to the FSP process (see Figure 6-10, Figure 6-11, and Figure 6-18).

b. Output signal *decoding started on A* sent to the FSP process (see Figure 6-19).

3.2.1.2.2 Wakeup symbol

The node shall support a dedicated wakeup symbol (WUS) composed of *gdWakeupTxActive* bits transmitted at a LOW level followed by *gdWakeupTxIdle* bits of 'idle'. A node generates a wakeup pattern (WUP) by repeating the wakeup symbol *pWakeupPattern* times[33]. An example of a wakeup pattern formed by a sequence of two wakeup symbols is shown in Figure 3-5.

[32] The delay in TxD at the end of transmission (i.e., a stagger in the deactivation of the TxEN and TxD outputs) is done to avoid the possibility of momentary glitches at the end of symbol transmission that could arise if systematic delays in CC, interface, or BD would cause the TxD signal to effectively transition before the TxEN signal is deactivated. By staggering the deactivation the possibility of such glitches is avoided. Note that a similar situation that could arise at the start of transmission requires no special treatment due to the behavioral characteristics of the BD, where a transmission does not start until both the TxEN and TxD inputs are active (see [EPL10] for further details).

[33] *pWakeupPattern* is a configurable parameter that indicates how many times the WUS is repeated to form a WUP. The required value of *pWakeupPattern* is affected by the number of active stars in the communication channel, and by the detailed wakeup forwarding characteristics of these active stars. Details of this configuration are beyond the scope of this document. Refer to [EPL10], [EPLAN10], and the documentation of the active star components for further details.

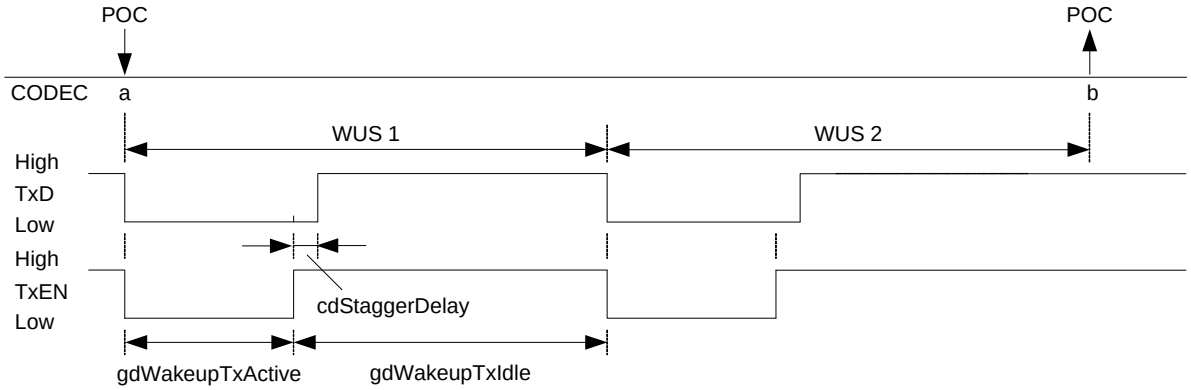

Figure 3-5: Wakeup pattern consisting of two wakeup symbols.

Figure 3-5 shows the bit stream of a wakeup pattern and related events relevant to the CODEC process:

a. Input signal *transmit symbol on A (vCEType)* received from the POC process with *vCEType* = WUP (see Figure 7-4).

b. Output signal *WUP transmitted on A* sent to the POC process (see Figure 7-4).

The node shall transmit a WUS with the edges of the TxEN signal being synchronous to the TxD at the start of the WUS's low phase, and with TxD returning to high after a delay period (defined by the parameter *cdStaggerDelay*) following the point that the TxEN signal returns to high at the start of the WUS's idle phase (which is also the end of the WUS's low phase)[34]. Note that there is no TSS transmission associated with a WUS. The node must be capable of detecting activity on the channel during *gdWakeupTxIdle* inside a WUP as shown in Figure 3-6.

[34] The purpose of the staggering of the TxD and TxEN is the same as for transmission of the CAS/MTS - to avoid the possibility of glitches when the TxEN output is deactivated.

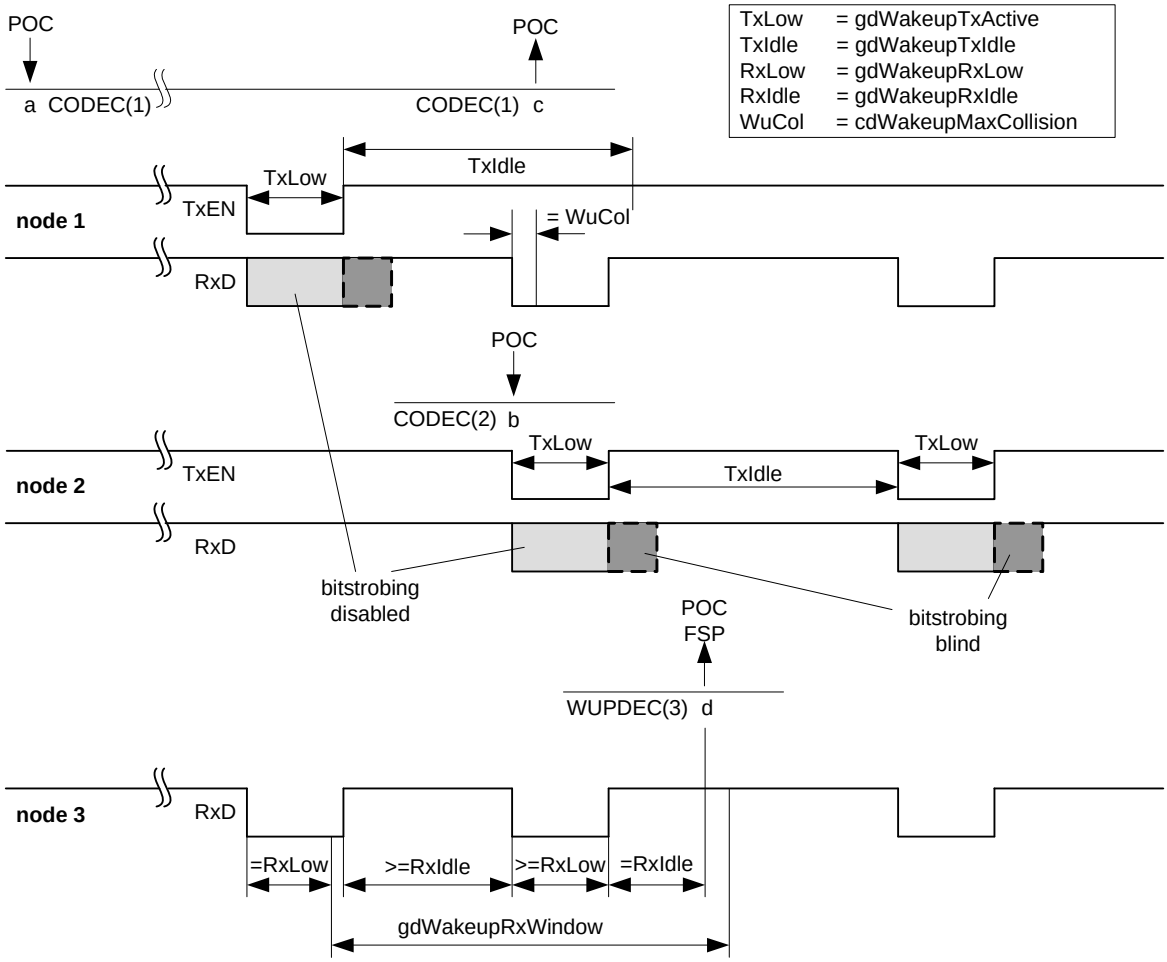

Figure 3-6: Wakeup symbol collision and wakeup pattern reception.

Figure 3-6 shows an example bit stream that could result from a wakeup symbol collision and shows related events relevant to the CODEC and WUPDEC processes:[35]

a. Input signal *transmit symbol on A (vCEType)* received from the POC process of node 1 with *vCEType* = WUP (see Figure 7-4).

b. Input signal *transmit symbol on A (vCEType)* received from the POC process of node 2 with *vCEType* = WUP (see Figure 7-4).

c. Output signal *wakeup collision on A* sent from the CODEC process of node 1 to the POC process of node 1 (see Figure 7-4).

d. Output signal *wakeup decoded on A* sent from the WUPDEC process of node 3 to the POC process (see 3.2.7.2.2) and to the FSP process (see Figure 6-8).

[35] In the figure, node 2 transmits a WUP even though node 1 had previously begun transmitting a WUP. This can occur if node 2 does not receive the WUS's previously sent by node 1. One possible reason that this could occur is that the first several WUS's of the wakeup pattern were used to wake up a sleeping star positioned between node 1 and node 2. On the other hand, node 3 may receive more of the WUS's sent by node 1 if the path between node 1 and node 3 consumes fewer WUS's than the path between node 1 and node 2.

3.2.1.2.3 Wakeup During Operation Pattern (WUDOP)

The node shall support the transmission of a Wakeup During Operation Pattern (WUDOP), intended to allow the node to send a pattern during normal operation that will cause a remote wakeup-capable BD that is in the low power state to detect a wakeup. The WUDOP consists of a sequence of LOW-HIGH-LOW-HIGH-LOW phases followed by a brief HIGH phase with a duration of a single bit[36]. The durations of all of the phases except for the last are *gdWakeupTxActive* bits.

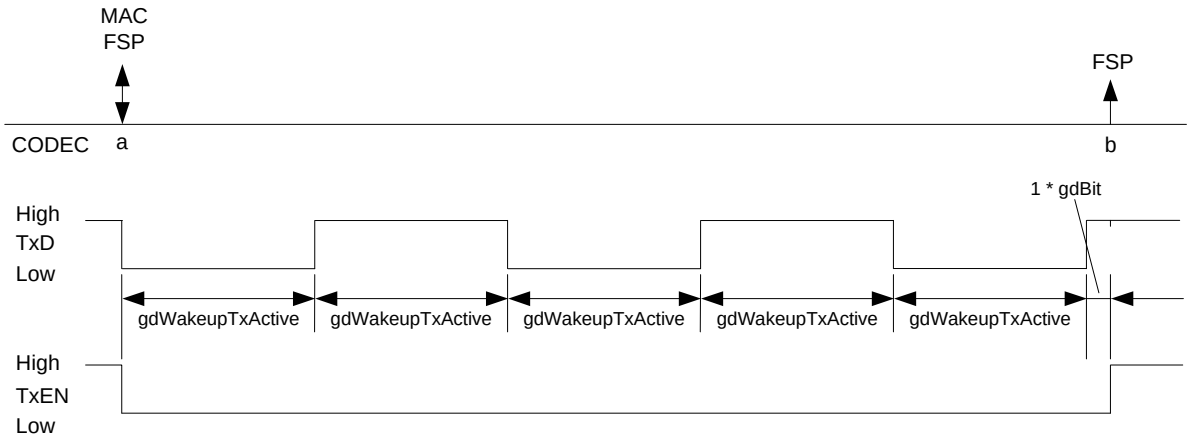

Figure 3-7: Wakeup During Operation Pattern.

Figure 3-7 shows the bit stream of a WUDOP and related events relevant to the CODEC process:

a. Input signal *transmit symbol on A* (*vCEType*) received from the MAC process with *vCEType* = WUDOP (see Figure 5-26) and output signal *decoding halted on A* sent to the FSP process (see Figure 6-10, Figure 6-11, and Figure 6-18).

b. Output signal *decoding started on A* sent to the FSP process (see Figure 6-19).

The node shall transmit a WUDOP with the leading edge of the TxEN signal synchronous to the TxD at the start of the WUDOP's first low phase, and with TxD returning to high one bit time before the TxEN signal returns to high at the end of the WUDOP. Note that there is no TSS transmission associated with a WUDOP. Also note that before and after the actively transmitted WUDOP, a receiver must see phases of idle on the bus. This is achieved by an appropriate configuration of the symbol window action point (see section B.4.10) and the size of the symbol window (see section B.4.13).

3.2.2 Sampling and majority voting

The node shall perform sampling on the RxD input, i.e., for each channel sample clock period the node shall sample and store the level of the RxD input[37]. The node shall temporarily store the most recent *cVotingSamples* samples of the input.

The node shall perform a majority voting operation on the sampled RxD signal. The purpose of the majority voting operation is to filter the RxD signal (the sampled RxD signal is the input and a *voted RxD on A* signal is the output). The majority voting mechanism is a filter for suppressing glitches (spikes) on the RxD input signal. In the context of this chapter a glitch is defined to be an event that changes the current condition of the physical layer such that its detected logic state is temporarily forced to a value different than what is being sent on the channel by the transmitting node.

[36] The single bit of high at the end of the WUDOP is required for idle detection stability of certain types of physical layers.

[37] CC's that support two channels shall perform the sampling and majority voting operations for both channels. The channels are independent (i.e., the mechanism results in two sets of voted values, one for channel A and another for channel B).

The decoder shall continuously evaluate the last stored *cVotingSamples* samples (i.e., the samples that are within the majority voting window) and shall calculate the number of HIGH samples. If the majority of the samples are HIGH then the voting unit output signal *zVotedVal* is HIGH, otherwise *zVotedVal* is LOW. The parameter *zVotedVal* captures the current value of the *voted RxD on A* signal as depicted within the BITSTRB process in Figure 3-38.

Figure 3-8 depicts a sampling and majority voting example. A rising edge on the channel sample clock causes the current value from the RxD bit stream to be sampled and stored within a stabilizing flip-flop. The next rising edge on the sample clock shifts the value from the stabilizing flip-flop into the voting window. The majority of samples within the voting window determines the *zVotedVal* output; the level of *zVotedVal* changes as this majority changes. Single glitches that affect only one or two channel sample clock periods are suppressed. In the absence of glitches, the value of *zVotedVal* has a fixed delay of *cVotingDelay* + *adInternalRxDelay*[38] sample clock periods relative to the value of the sampled RxD.

All other mechanisms of the decoding process shall consume the signal *bit strobed on A (zVotedVal)* generated by the BITSTRB process (see Figure 3-38) and shall not directly consider the RxD serial data input signal.

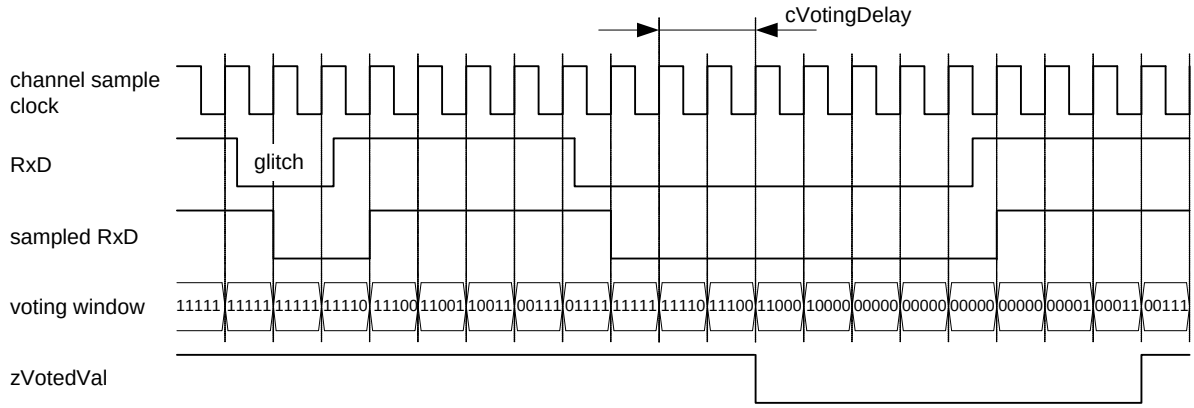

Figure 3-8: Sampling and majority voting of the RxD input (*adInternalRxDelay* = 1)**.**

3.2.3 Bit clock alignment and bit strobing

The bit clock alignment mechanism synchronizes the local bit clock used for bit strobing to the received bit stream represented by the *zVotedVal* variable.

A sample counter shall count the samples of *zVotedVal* cyclically in the range of 1 to *cSamplesPerBit*.

A bit synchronization edge is used to realign the bit timing of the receiver (i.e., bit clock resynchronization). The node shall enable the bit synchronization edge detection each time a HIGH bit is strobed except when a HIGH bit is strobed while decoding bits from a byte in the header, payload or trailer. Synchronization is enabled for the edge between the two bits in the BSS for these bytes, however.

The bit clock alignment shall perform the bit synchronization when it is enabled and when *zVotedVal* changes to LOW.

When a bit synchronization edge is detected (and bit synchronization is enabled) the bit clock alignment shall not increment the sample counter but instead shall set it to a value of two for the next sample. The node shall only perform bit synchronization on HIGH to LOW transitions of *zVotedVal* (i.e., on the falling edge of the majority voted samples)[39].

[38] In this example the optional stabilizing flip-flop introduces the required minimum internal delay (*adInternalRxDelay*) of one sample up to the strobe point. Note that an additional delay after the determination of *zVotedVal* is possible, this delay would then increase the value of *adInternalRxDelay*.

[39] This is necessary as the output of the physical layer may have different rise and fall times for rising and falling edges.

Whenever a bit synchronization is performed, the bit clock alignment mechanism shall disable further bit synchronizations until it is enabled again as described above. The bit stream decoding process shall perform at most one bit synchronization between any two consecutive bit strobe points.

The bit synchronization mechanism defines the phase of the cyclic sample counter, which in turn determines the position of the strobe point. The strobe point is the point in time when the cyclic sample counter value is equal to *cStrobeOffset*.

A bit shall be strobed when the cyclic sample counter is at the value *cStrobeOffset*, if this does not coincide with a bit synchronization. When this condition is fulfilled, the current value of *zVotedVal* is taken as the current bit value and is signalled to the other processes. This action is called bit strobing.

Figure 3-9 shows the mechanism of the bit synchronization when a frame is received.

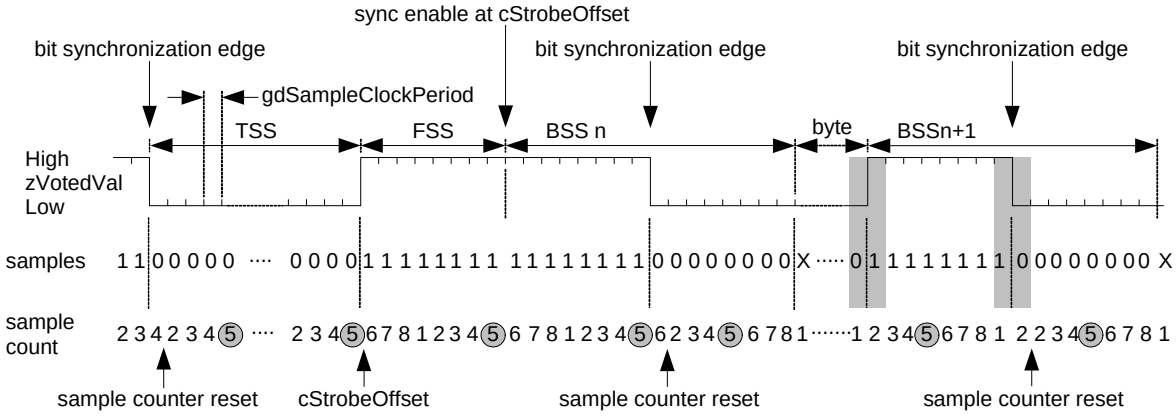

Figure 3-9: Bit synchronization.

The example from Figure 3-9 shows the nominal case of an FSS and BSS with *cSamplesPerBit* samples. At the bit synchronization edge, the sample counter is set to '2' for the sample following the detected edge. The example also shows a misalignment after the received first header byte of the frame. The misalignment is reflected by the value of the sample counter at the start of the HIGH bit of the BSS (see the first highlighted area). The first expected sample (HIGH) of BSS n + 1 should occur when the sample counter value is '1'. Since it actually occurs when the sample counter value is '2', the edge was decoded with a delay of one channel sample clock period. Bit synchronization, performed by resetting the sample counter, takes place with the next bit synchronization edge (see second highlighted area). The effect of the bit synchronization is that the distance of edge to the strobe point is the same as if the edge would have appeared at the expected sample (see also 3.2.8).

To detect activity on a channel, it is necessary that at least *cStrobeOffset* consecutive LOW samples make it through the majority voter. This is a consequence of the combination of the majority voting and the bit synchronization mechanisms.

The SDL representation of the BITSTRB process is depicted in Figure 3-38.

3.2.4 Implementation specific delays

In addition to the delays on the path from RxD input to the signal *bit strobed on A (zVotedVal)* implied by the definition of the voting mechanism (*cVotingDelay*) and the bit strobing mechanism (*cStrobeOffset*) an actual implementation might need to add an additional delay (*adInternalRxDelay*) into this path.

Such an additional delay might for example be the sum of delays introduced by stabilizing flip-flops after the RXD input or an additional flip-flop after the voting mechanism. The FlexRay protocol specification tolerates such an additional delay in the range from *cdInternalRxDelayMin* to *cdInternalRxDelayMax* sample clock periods (see B.4.2.1.1). The important conformance test criteria is that the overall delay in this path is in the range defined by the sum of *cVotingDelay* + *cStrobeOffset* + *adInternalRxDelay*.

If an implementation introduces an additional delay greater than *cdInternalRxDelayMax* or smaller than *cdInternalRxDelayMin* it internally has to compensate the delay.

3.2.5 Channel idle detection

The node shall use a channel idle detection mechanism to identify the end of the current communication element. Idle detection is done by means of a sample tick timer which is set to a duration of *cChannelIdleDelimiter* times *cSamplesPerBit*. This idle timer is started (or restarted) whenever a bit is strobed as low by the BITSTRB process - expiration of the timer (which implies that no bits have been strobed as low during the duration of the timer[40]) results in the detection of idle (i.e., the channel is considered to be idle). The channel continues to be considered idle until a bit is strobed as low, which causes the channel to be considered to be active and once again starts the idle timer. Channel idle detection is not active while the node is encoding a communication element - the decoding and encoding mechanisms are mutually exclusive and idle detection is a logical component of the decoding mechanism.

When the CODEC process is instantiated by the POC, the CODEC process immediately instantiates the BITSTRB process and puts it in the GO mode where idle detection is performed. The initial assumption is that the channel is active, so *cChannelIdleDelimiter* times *cSamplesPerBit* sample ticks must go by without strobing a low bit before the channel is considered to be idle (see section 3.4.2).

When the CODEC process is encoding a communication element, the BITSTRB process is placed in the STANDBY mode and idle detection is stopped. Following the completion of the encoding of a communication element, a mode control signal is sent to the BITSTRB process that causes the process to restart the idle timer and once again begin idle detection.

Because of certain effects on the physical layer, a node that has just completed a transmission may experience a period of time where the signal seen at the RxD input does not reflect what is actually occurring on the bus. For example, for a period of time following the completion of a transmission the RxD input may indicate LOW even though no node in the system is actively driving the bus. In order to overcome this issue, when the BITSTRB process is restarted following a transmission it is placed into the BLIND mode. While operating in this mode, the BITSTRB process basically ignores the actual status of the RxD input and acts as if the RxD input was indicating HIGH at all times. The BITSTRB process remains in the BLIND mode for a configurable number of bit times (determined by the parameter *gdIgnoreAfterTx*) after which it returns to the GO mode where bits are strobed in the normal manner. Although no actual bits are strobed while the BITSTRB process is in the BLIND mode, when BITSTRB leaves the BLIND mode it behaves as if it had strobed *gdIgnoreAfterTx* consecutive HIGH bits.

3.2.6 Action point and time reference point

As defined in Chapter 5, an action point (AP) is an instant in time at which a node performs a specific action in alignment with its local time base, e.g. when a transmitter starts the transmission of a frame.

The clock synchronization algorithm requires a measurement of the time difference between the static slot action point of the transmitter of a sync frame and the static slot action point of the corresponding slot in the receiving node. Obviously, a receiving node does not have direct knowledge of the static slot action point of a different node. The clock synchronization algorithm instead infers the time of the transmitter's action point by making a measurement of the arrival time of a received sync frame[41].

[40] This is not necessarily equivalent to strobing *cChannelIdleDelimiter* consecutive high bits because under some circumstances the strobe point can be modified by the bit clock alignment mechanism discussed in section 3.2.3.

[41] This is possible because transmission of the sync frame begins at the static slot action point of the transmitting node.

Due to certain effects on the physical transmission medium it is possible that the first edge at the start of a frame is delayed different than all other edges of the same frame, causing the TSS seen at the RxD input to be shorter or longer than the TSS that was transmitted. This effect is called TSS length change and it has various causes (e.g., connection setup in active stars, differences in propagation delays of rising and falling edges, etc.). The cumulative effect of all such causes on a TSS transmitted from node M to node N is to change the length of the TSS by *dFrameTSSLengthChange*$_{M,N}$. A node shall accept the TSS as valid if any number of consecutive strobed logical LOW bits in the range of 1 to (*gdTSSTransmitter* + 2) is detected.[42]

Signals transmitted from a node M are received at node N with the propagation delay *dPropagation-Delay*$_{M,N}$. The propagation delay is considered to be the same for all corresponding edges in the transmit TxD signal of node M to the receive RxD signal at node N except for the first edge at the start of the frame. Figure 3-10 depicts the effect of propagation delay and TSS length change. For a more detailed description refer to [EPL10].

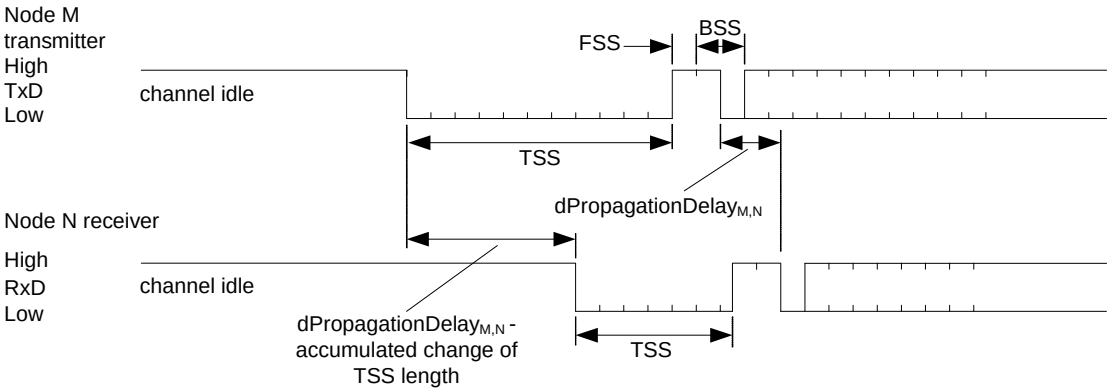

Figure 3-10: TSS length change and propagation.

As a result of TSS length change and propagation delay, it is not possible to know the precise relationship between when a receiver begins to see a TSS and when the transmitter started to send the TSS. It is necessary to base the time measurements of received frames on an element of the frame that is not affected by TSS length change. The receiving node takes the timestamp of a secondary time reference point (TRP) that occurs during the first BSS of a message and uses this to calculate the timestamp of a primary TRP that represents when the node should have seen the start of the TSS if the TSS had not been affected by TSS length change and propagation delay. The timestamp of the primary TRP is used as the observed arrival time of the frame by the clock synchronization algorithm.

The strobe point of the second bit of the first BSS in a frame (i.e., the first HIGH to LOW edge detected after a valid TSS) is defined to be the secondary TRP. A receiver shall capture a time stamp, *zSecondaryTRP*, at the secondary TRP of each potential frame start.

The node shall calculate a primary TRP, *zPrimaryTRP*, from the secondary TRP timestamp. The *zPrimaryTRP* timestamp serves as the sync frame's observed arrival time for the clock sync, and is passed onto the FSP process via the *frame decoded on A (vRF)* signal. Both *zPrimaryTRP* and *zSecondaryTRP* are measured in microticks.

[42] The use of the "+ 2" constant places requirements on the EPL parameters such as *dFrameTSSLengthChange* and *dFrameTSSEMIInfluence*. See Table B-5 for additional details.

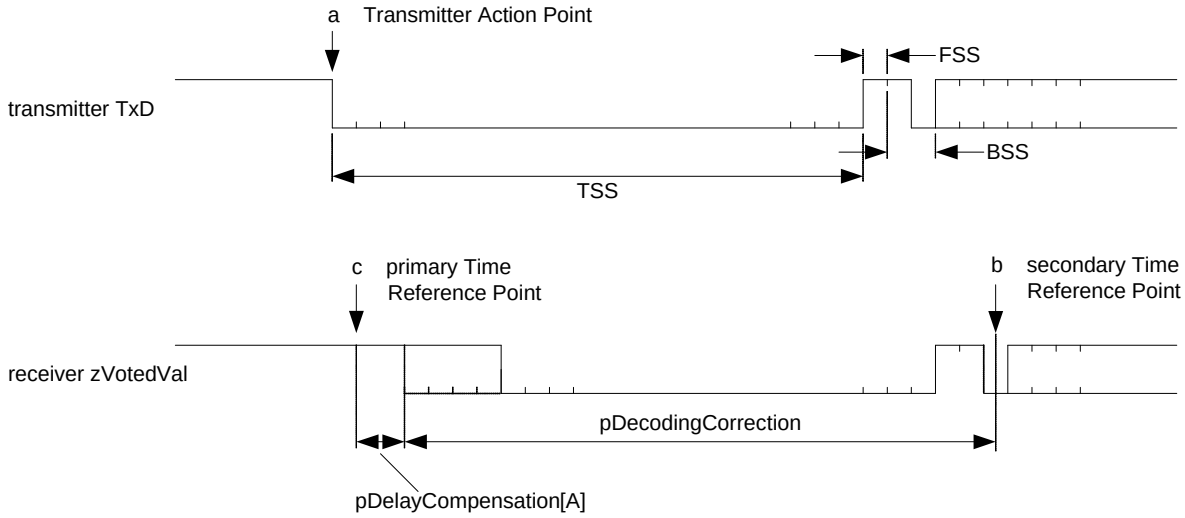

Figure 3-11: Time reference point definitions.

Figure 3-11 depicts definitions for the time reference point calculations and shows the following significant events:

a. Transmitter static slot action point - the point at which the transmitter begins sending its frame

b. Secondary TRP (timestamp *zSecondaryTRP*), located at the strobe point of the second bit of the first BSS. At this point in time, the decoding process shall provide the output signal *potential frame start on A* to the CSS on channel A process (see also section 8.4.2).

c. Primary TRP (timestamp *zPrimaryTRP*), calculated from *zSecondaryTRP* by subtracting a fixed offset (*pDecodingCorrection*) and a delay compensation term (*pDelayCompensation*) that attempts to correct for the effects of propagation delay. Note that this does not represent an actual event, but rather only indicates the point in time that the timestamp represents.

The difference between *zPrimaryTRP* and *zSecondaryTRP* is the summation of node parameters *pDecodingCorrection* and *pDelayCompensation*. The calculation of *pDecodingCorrection* is given in B.4.25.

The Primary TRP timestamp is passed to the FSP process (and subsequently to the clock synchronization process) via the PrimaryTRP element of the *vRF* structure (see Figure 3-36, Figure 6-9, Figure 6-11, and Figure 8-14). The clock synchronization algorithm uses the deviation between *zPrimaryTRP* and the sync frame's expected arrival time to calculate and compensate the node's local clock deviation. For additional details concerning the time difference measurements see Chapter 8.

3.2.7 Frame and symbol decoding

This section specifies the mechanisms used to perform bit stream decoding. The decoding part of the CODEC process interprets the bit stream observed at the voted RxD input of the node (provided by the sequence of *bit strobed on A* signals from the BITSTRB process).

The decoding part of the CODEC process does not perform concurrent decoding of frames and CAS/MTS symbols, i.e. for a given channel the CODEC process shall support only one decoding method (frame or CAS/MTS symbol) at a time. For example, once a new communication element is classified as a start of a frame then CAS/MTS decoding is not required until the channel has been detected as idle again after the end of the frame and/or after a decoding error is detected.

In addition to the CODEC process the WUPDEC process is permanently performing wakeup pattern detection unless the CODEC is transmitting a CE or is in STANDBY. This detection is active even if the CODEC is currently decoding a frame or a CAS/MTS symbol.

The decoding process shall support successful decoding[43] of consecutive communication elements when the spacing between the last bit of the previous element and the first bit of the subsequent communication element is greater than or equal to *cChannelIdleDelimiter* bits.

The bit stream decoding of the individual channels on a dual channel node shall operate independently from one another.

The node shall derive the channel sample clock period *gdSampleClockPeriod* from the oscillator clock period directly or by means of division or multiplication. In addition to the channel sample clock period, the decoding process shall operate based on the programmed bit length as characterized by the parameter *gdBit*. The programmed bit length is an integer multiple of the channel sample clock period. It is defined to be the product of samples per bit *cSamplesPerBit* and the channel sample clock period *gdSampleClock-Period*.

The relation between the channel sample clock period and the microtick is characterized by the microtick prescaler *pSamplesPerMicrotick*. The channel sample clock and the microtick clock must be synchronized, i.e., there must be an integer multiplicative relationship between the two periods and the two clocks must have a fixed phase relationship.

3.2.7.1 Frame decoding

A frame starts at CE start with the first strobed LOW bit after channel idle. The channel idle delimiter refers to the time required by the idle detection mechanism to determine that the channel is idle (refer to section 3.2.5 for details). In order for idle to occur all bits strobed during the channel idle delimiter must be strobed as high.

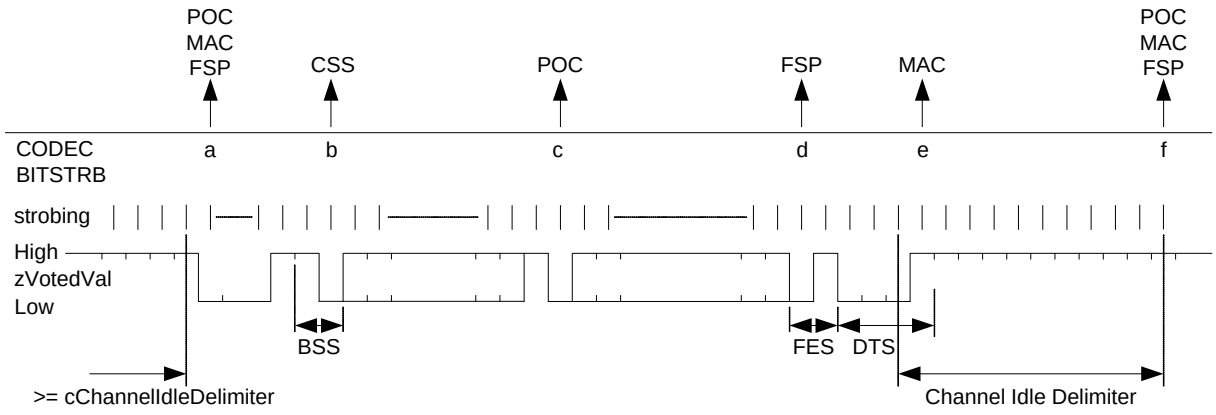

Figure 3-12: Received frame bit stream.

Figure 3-12 shows the received bit stream of a frame and events in relation to the CODEC and the BITSTRB processes:

a. Output signal *idle end on A* to the POC process shown in Figure 2-13, Figure 7-3, and Figure 7-14, and to the MAC process shown in Figure 5-20;
output signal *CE start on A* to the MAC process shown in Figure 5-21, and to the FSP process shown in Figure 6-10.

b. Output signal *potential frame start on A* to the CSS process shown in Figure 8-11.

c. Output signal *header received on A* to the POC process shown in Figure 7-3, Figure 7-5, Figure 7-14, Figure 7-16, and Figure 7-18.

d. Output signal *frame decoded on A (vRF)* to the FSP process shown in Figure 6-11.

[43] Note that successful decoding does not necessarily imply successful reception in terms of being able to present the payload of the decoded stream to the host.

e. Output signal *potential idle start on A* to the MAC process shown in Figure 5-21; Output signal *DTS received on A* to the MAC process shown in Figure 5-21.

f. Output signal *CHIRP on A* to the MAC process shown in Figure 5-21 and Figure 5-20, to the FSP process shown in Figure 6-18, and to the POC process shown in Figure 2-13, Figure 7-3, and Figure 7-14.

Note that the BITSTRB process will output a *potential idle start on A* signal every time a bit strobed as high was preceded by a bit strobed as low. To keep the figure simple these signals are not shown in Figure 3-12 with the exception of the potential idle start at the end of the DTS. Although all of these *potential idle start* signals are processed by the MAC process, only the one associated with the end of the DTS is relevant to the operation of dynamic segment media access.

3.2.7.2 Symbol decoding

3.2.7.2.1 Collision avoidance symbol and media access test symbol decoding

The node shall decode the CAS and MTS symbols in exactly the same manner. Since these symbols are encoded by a LOW level of duration *cdCAS* starting immediately after the TSS, it is not possible for receivers to detect the boundary between the TSS and the subsequent LOW bits that make up the CAS or MTS.

As a result, the detection of a CAS or MTS shall be considered as valid coding if a LOW level with a duration between *cdCASRxLowMin* and *gdCASRxLowMax* is detected.

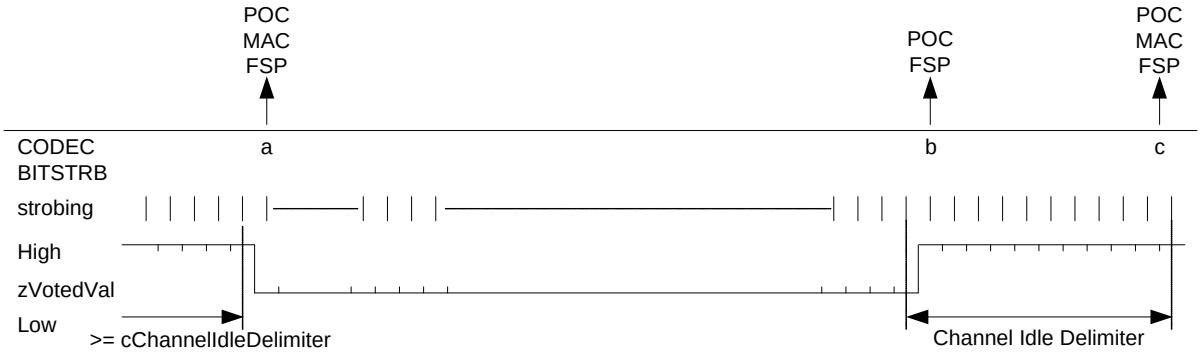

Figure 3-13: Received CAS/MTS bit stream.

Figure 3-13 shows the received bit stream of a CAS/MTS and events in relation to the CODEC and the BITSTRB processes:

a. Output signal *idle end on A* to the POC process shown in Figure 2-13, Figure 7-3, and Figure 7-14, and to the MAC process shown in Figure 5-20;
output signal *CE start on A* to the MAC process shown in Figure 5-21, and to the FSP process shown in Figure 6-10.

b. Output signal *CAS_MTS decoded on A* to the POC process shown in Figure 7-3, Figure 7-14, Figure 7-16, and Figure 7-18, to the MAC process shown in Figure 5-21, and to the FSP process shown in Figure 6-12.

c. Output signal *CHIRP on A* to the POC process shown in Figure 2-13, Figure 7-3, and Figure 7-14, to the MAC process shown in Figure 5-21 and Figure 5-20, and to the FSP process shown in Figure 6-18.

3.2.7.2.2 Wakeup symbol decoding

The detection of either a WUDOP or a WUP composed of at least 2 WUS's shall be considered as valid coding if all of the following conditions are met:

1. A HIGH level with a duration of at least *gdWakeupRxIdle* is detected.

2. This is followed by a duration of at least *gdWakeupRxLow* at the LOW level. After this duration a window timer is started, even if the LOW level is still present. The timer will expire after *gdWakeupRxWindow*.
3. This is followed by a duration of at least *gdWakeupRxIdle* at the HIGH level.
4. This is followed by a duration of at least *gdWakeupRxLow* at the LOW level.
5. This is followed by a duration of at least *gdWakeupRxIdle* at the HIGH level.
6. The last three phases of HIGH, LOW and HIGH are received before the window timer that was started during the first LOW phase has expired.

The bit stream received by node 3 in Figure 3-6 shows the reception of a WUP and the related event relevant to the WUPDEC process of node 3:

d. Output signal *wakeup decoded on A* to the POC process shown in Figure 7-3 and Figure 7-5, and to the FSP process shown in Figure 6-8.

3.2.7.3 Decoding error

Exiting one of the decoding macros CAS_MTS_DECODING, FSS_BSS_DECODING, HEADER_DECODING, PAYLOAD_DECODING, or TRAILER_DECODING (shown in Figure 3-26) with an exit condition of decoding error shall abort and restart the decoding again. Prior to doing so, the FSP is informed about the decoding error.

When a decoding error is detected the node shall treat the first wrong bit as the last bit of the current communication element, i.e. it shall terminate communication element decoding and shall wait for successful channel idle detection.

The following condition shall not lead to a decoding error:

- One or three HIGH bits (instead of exactly two HIGH bits) are strobed after the TSS.

In the FSS-BSS sequence, it is possible that, due to the quantization error of one sample period from the receiver's point of view, an incoming HIGH level of 2 gdBit length (FSS + first bit of the BSS) may be interpreted as

1. (2 * *cSamplesPerBit* - 1) or
2. (2 * *cSamplesPerBit*) or
3. (2 * *cSamplesPerBit* + 1)

samples long.

This could also arise as a systematic effect due to slow/fast clocks of the node and the analog-to-digital conversion of the signal.

Figure 3-14 shows an example FSS-BSS decoding with only (2 * *cSamplesPerBit* - 1) samples of HIGH. Under all conditions at least one leading HIGH bit between the TSS and the first data byte is accurately decoded.

In addition to the effects of quantization, the presence of asymmetry could also result in additional reduction or lengthening of the actual duration of the HIGH period of the FSS-BSS sequence. Any amount of asymmetry that is accommodated by the decoding algorithm would still result in the strobing of between one and three bits at HIGH. Refer to [EPLAN10] for additional details.

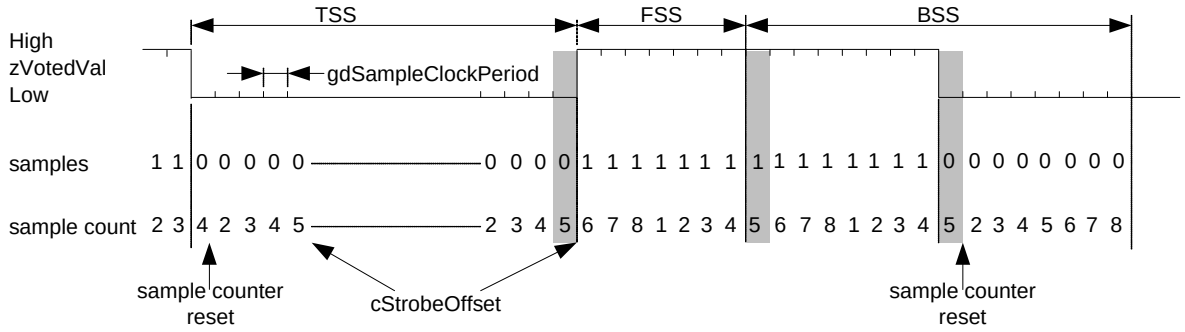

Figure 3-14: Start of frame with FSS BSS decoding.

3.2.8 Signal integrity

In general, there are several conditions (e.g. clock oscillator differences, electrical characteristics of the transmission media or the transceivers, EMI etc.) that can cause variations of signal timing or introduce anomalies/glitches into the communication bit stream. The decoding function attempts to enable tolerance of the physical layer against presence of one glitch in a bit cell when the length of the glitch is less than or equal to one channel sample clock period.[44]

Asymmetric delays cause bit edges to occur earlier or later than would otherwise be expected from a receiver's perspective. This could contribute to individual receivers incorrectly determining a received bit value. To avoid this effect these delays must be bounded such that the aggregate effect of asymmetric delays at any receiving node does not affect the majority of bit samples used in determining the voted value at the strobe offset. Asymmetry causes and effects are described and characterized in [EPL10].

3.3 Coding and decoding process

3.3.1 Operating modes

The POC shall set the operating mode of the CODEC for each communication channel. Definition 3-1 shows the formal definition of the CODEC operating modes.

```
newtype T_CodecMode
      literals STANDBY, NORMAL, READY;
endnewtype;
```

Definition 3-1: Formal definition of T_CodecMode.

1. In the STANDBY mode, the execution of the CODEC and all of its subprocesses are effectively halted.
2. In the READY mode the bit strobe process BITSTRB and the wakeup detection process WUPDEC are executed but the CODEC process waits in its *CODEC:ready* state.
3. In the NORMAL mode the CODEC process, the bit strobe process BITSTRB and the wakeup detection process WUPDEC are executed.

3.3.2 Coding and decoding process behavior

This section contains the formalized specification of the CODEC control process. Figure 3-15, Figure 3-16 and Figure 3-17 depict the specification of the CODEC control process and the termination of the CODEC process. When the CODEC process receives the POC signal *terminate CODEC_A*, the CODEC sends termination signals to its subprocesses before terminating itself.

[44] There are specific cases where a single glitch cannot be tolerated and others where two glitches can be tolerated.

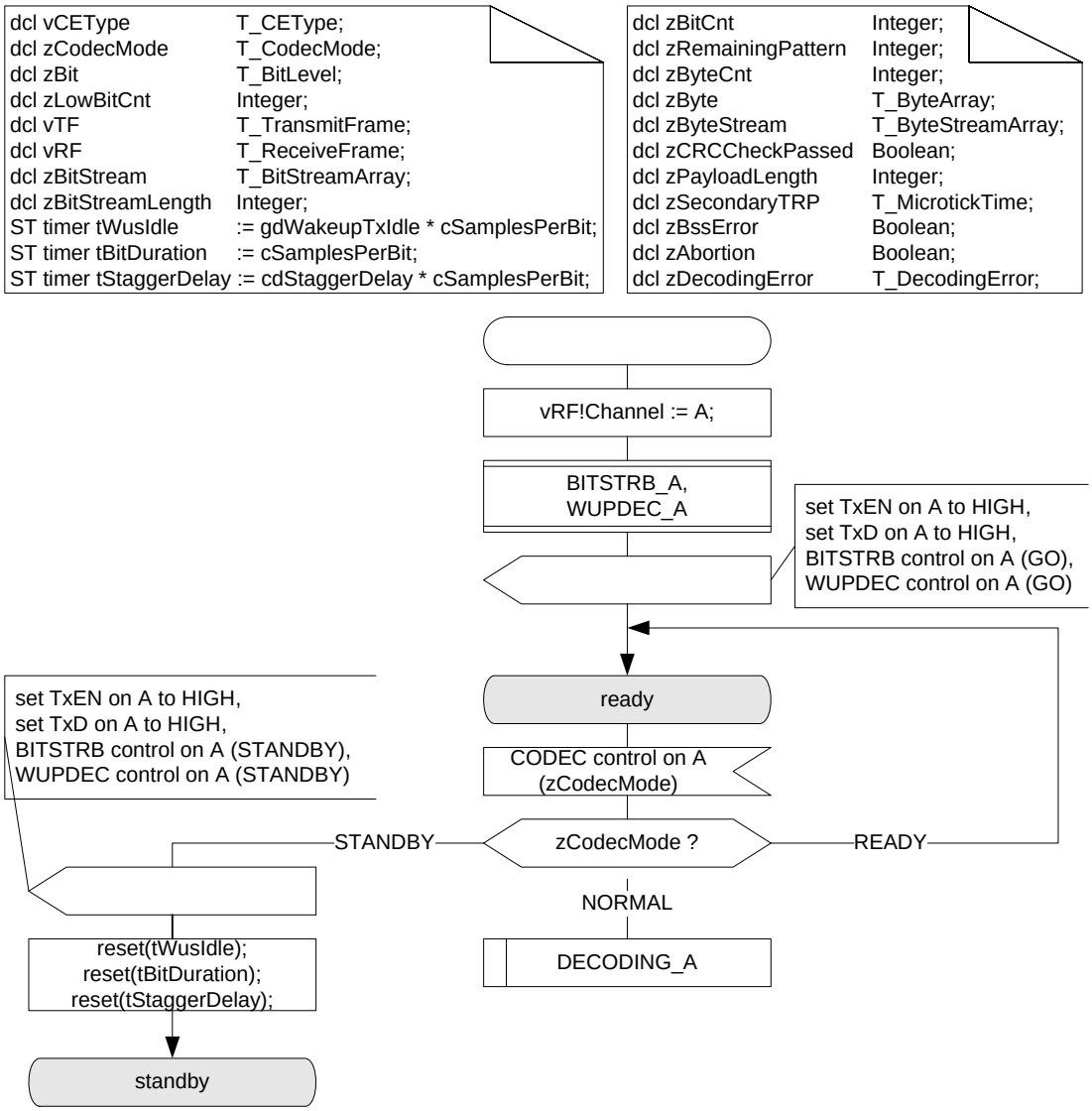

Figure 3-15: CODEC process [CODEC_A].

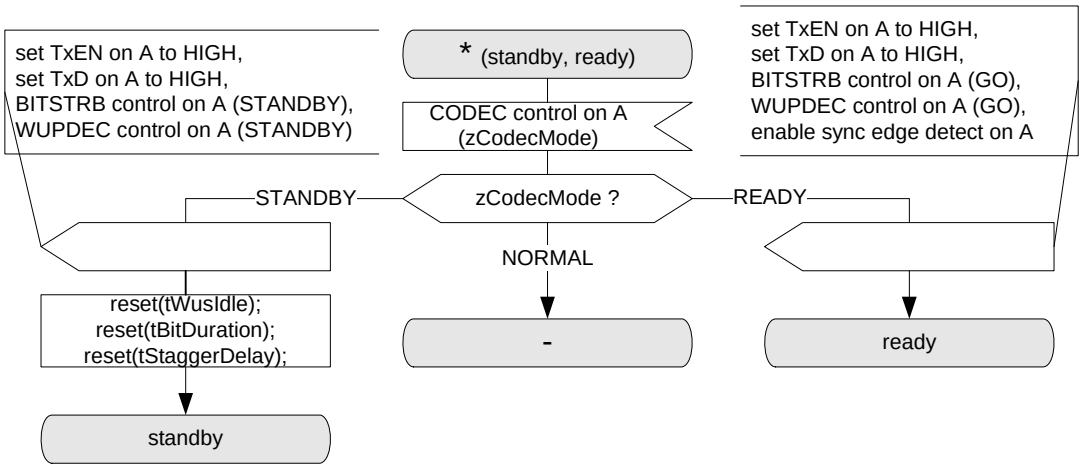

Figure 3-16: Mode control of the CODEC process [CODEC_A].

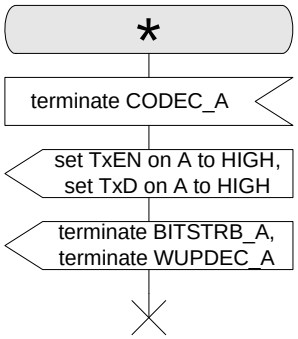

Figure 3-17: Termination of the CODEC process [CODEC_A].

3.3.3 Encoding behavior

The CODEC process receives the data to transmit from the media access control process. For frame transmission the variable *vTF* of type *T_TransmitFrame* is used, which is defined in Definition 3-2[45]:

```
newtype T_TransmitFrame
struct
      Header      T_Header;
      Payload     T_Payload;
endnewtype;
```

Definition 3-2: Formal definition of T_TransmitFrame.[46]

The variable *zBit* is used to describe the level of the TxD and TxEN interface signals between the CODEC and a bus driver. The *zBit* variable is of type *T_BitLevel* as defined in Definition 3-3:

```
newtype T_BitLevel
      literals HIGH, LOW;
```

[45] The frame sent on the channel also contains the frame CRC. The frame CRC is not part of the *vTF* variable - it is added to the frame by the CODEC.

[46] T_Header and T_Payload are defined in Chapter 4.

```
endnewtype;
```

Definition 3-3: Formal definition of T_BitLevel.

The CODEC process provides the assembled bit stream via the TxD signal to the BD (with the SDL signals *set TxD on A to HIGH* and *set TxD on A to LOW*) and controls the BD via the TxEN signal (the SDL signals *set TxEN on A to HIGH* and *set TxEN on A to LOW*). The transmitting node shall set TxD to HIGH in the case of a '1' bit and shall set TxD to LOW in the case of a '0' bit.

Figure 3-18 shows a high level view of the mechanisms used for encoding. After the reception of a *transmit frame on A* or *transmit symbol on A* signal the CODEC process follows one of four distinct paths depending on the type of communication element that is to be encoded. Each of these paths is described in a separate macro. Before any of the macros are executed, however, the CODEC process sets the TxEN output to LOW and puts the BITSTRB and WUPDEC processes into STANDBY mode.

If the CE to be encoded is a frame, the FRAME_ENCODING macro is executed (see Figure 3-19). This macro calls the **prepbitstream** function, which takes the inbound frame *vTF* from the MAC process, prepares the bit stream *zBitStream* of type *T_BitStreamArray* for transmission, and calculates the bit stream length *zBitStreamLength*. The **prepbitstream** function shall break the frame data down into individual bytes, prepend a TSS, add an FSS at the end of the TSS, create a series of extended byte sequences by adding a BSS at the beginning of each byte of frame data, calculate the frame CRC bytes and create the extended byte sequences for the CRC, and assemble a continuous bit stream out of the extended byte sequences. Once the bit stream is prepared, the macro steps bit by bit through the stream and presents it to the bus. Following this, the FRAME_ENCODING macro calls the TRANSMIT_FES macro and, if the frame is sent in the dynamic segment, the TRANSMIT_DTS macro. Once all of these are complete, the macro sends a control signal to the BITSTRB process that restarts bit strobing in the BLIND mode.

If the CE to be encoded is a CAS/MTS, the CAS_MTS_ENCODING macro is executed (see Figure 3-24). This macro simply sets the TxD output to LOW and leaves it at this state for an appropriate period of time (the duration of TSS plus the duration of the CAS). Once the necessary duration has passed (as indicated by the WAIT function described in Figure 3-22) the TxEN output is deactivated and the BITSTRB process is restarted in the BLIND mode. Following a delay of *cdStaggerDelay* bits (see section 3.2.1.2.1), the TxD output is also deactivated.

If the CE to be encoded is a Wakeup Pattern, the WUP_ENCODING macro is executed (see Figure 3-23). This macro loops through *pWakeupPattern* iterations, each one creating a WUS as described in section 3.2.1.2.2. As the low phase of each WUS is completed, the WUP_ENCODING macro provides the necessary staggering of the TxD and TxEN outputs, and commands the BITSTRB process to the BLIND mode to begin the process of allowing the detection of wakeup symbol collisions. If a collision is detected (by detecting *cdWakeupMaxCollision* consecutive LOW bits) the WUP_ENCODING macro is aborted, signaling a wakeup collision. If no collision is detected, the macro will loop until all WUS's have been generated and then exit.

If the CE to be encoded is a WUDOP, the WUDOP_ENCODING macro is executed (see Figure 3-25). This macro transmits a sequence of LOW-HIGH-LOW-HIGH-LOW phases followed by a brief HIGH phase. Following the last HIGH phase the TxEN output is deactivated and the BITSTRB process is restarted in the BLIND mode.

```
newtype T_BitStreamArray
      Array (Integer, T_BitLevel);
endnewtype;
```

Definition 3-4: Formal definition of T_BitStreamArray.

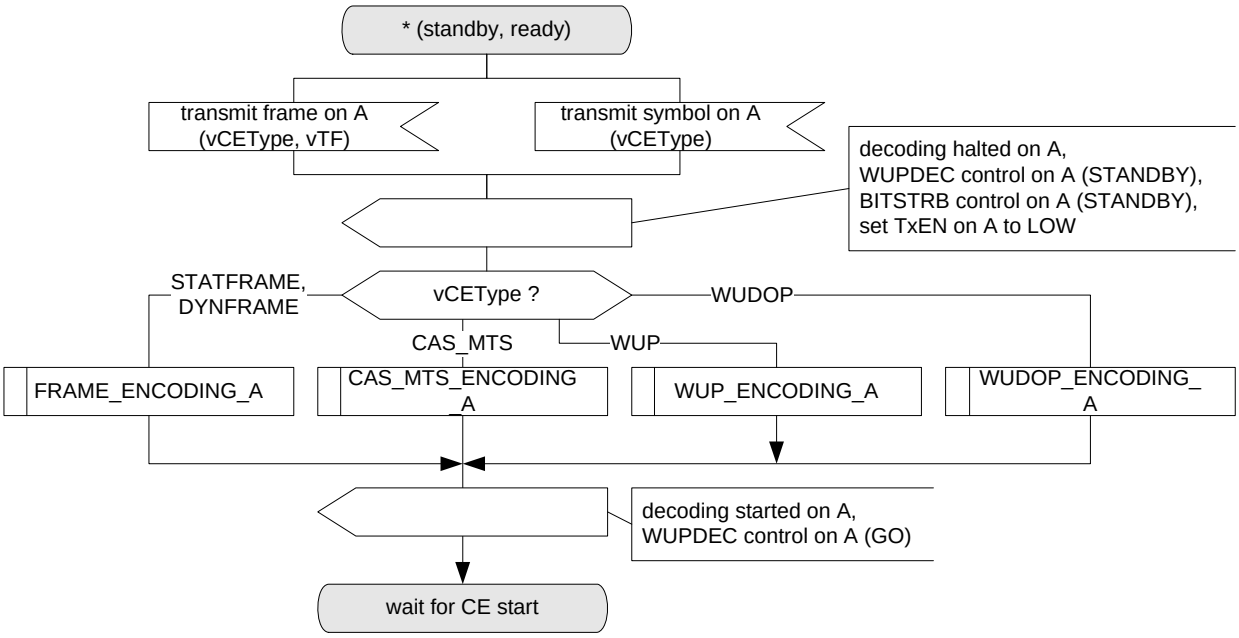

Figure 3-18: Encoding mechanism [CODEC_A].

Immediately after instantiating of the CODEC process, the encoder sends the signal *set TxEN on A to HIGH* to disable the BD's transmitter.

```
newtype T_CEType
      literals STATFRAME, DYNFRAME, CAS_MTS, WUP, WUDOP;
endnewtype;
```

Definition 3-5: Formal definition of T_CEType.

3.3.4 Encoding macros

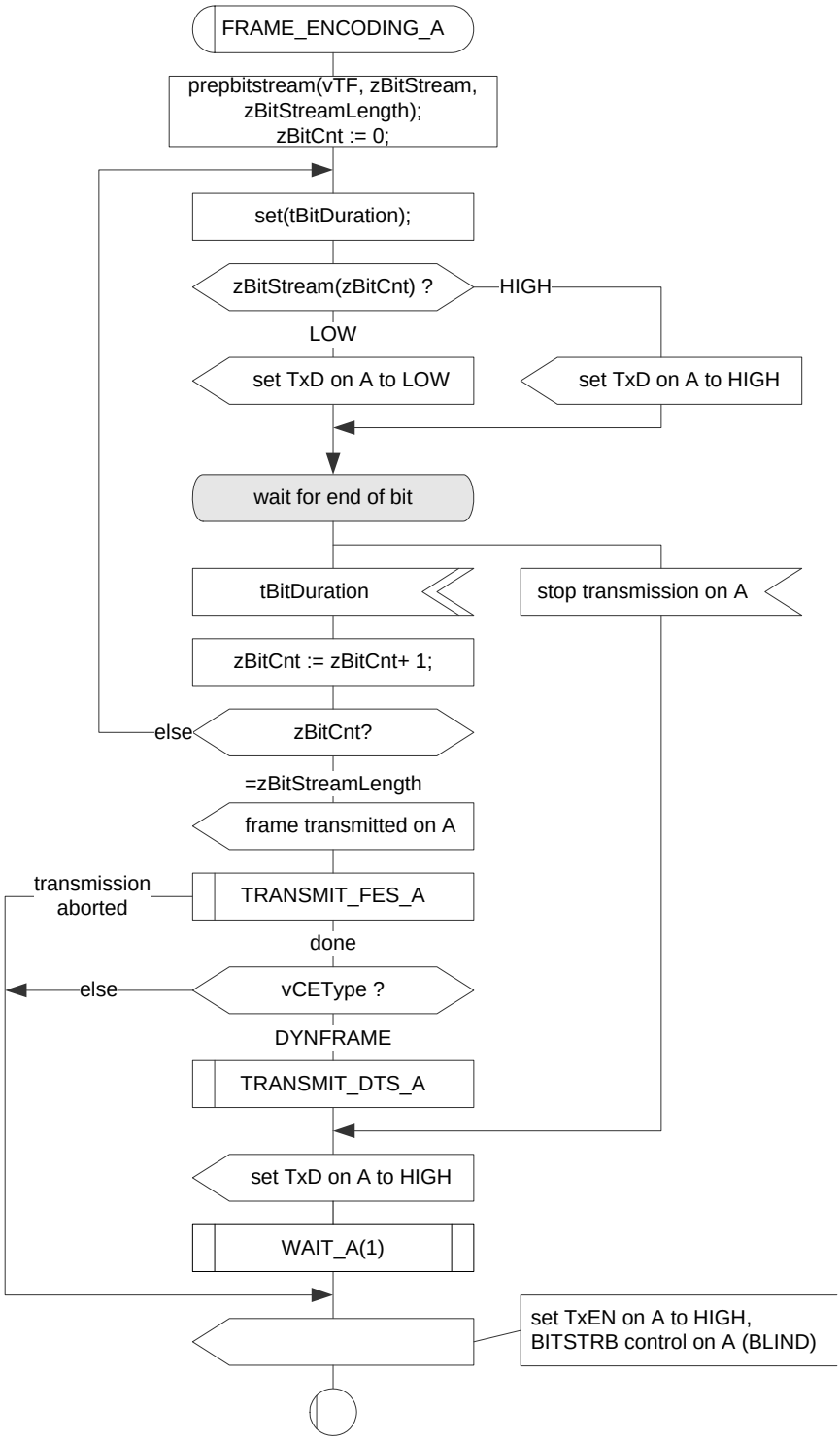

Figure 3-19: Encoding macro FRAME_ENCODING_A [CODEC_A].

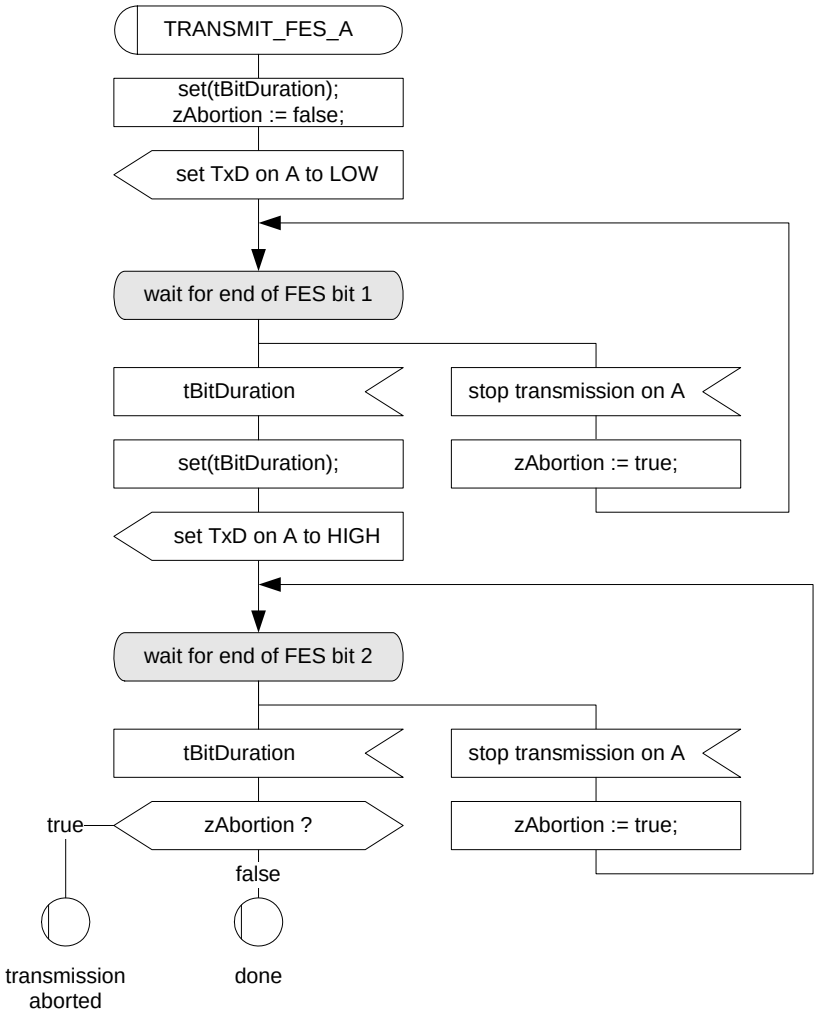

Figure 3-20: Encoding macro TRANSMIT_FES_A [CODEC_A].

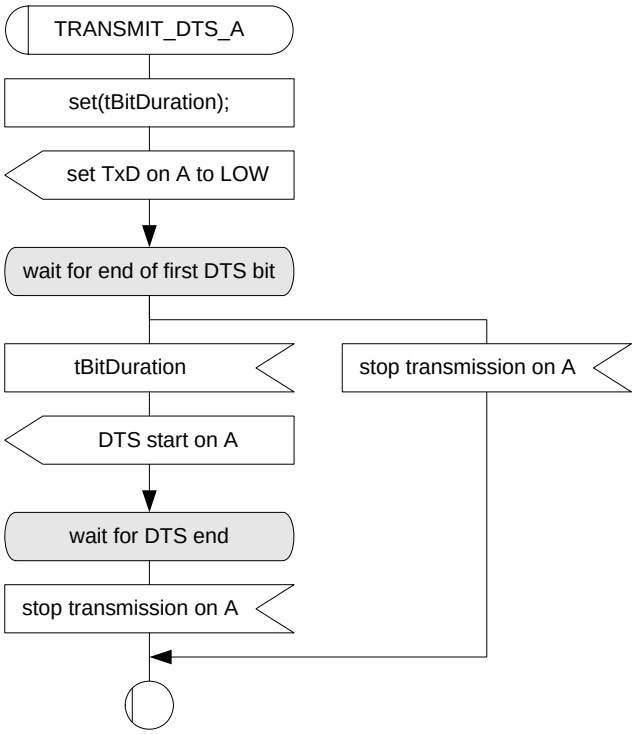

Figure 3-21: Encoding macro TRANSMIT_DTS_A [CODEC_A].

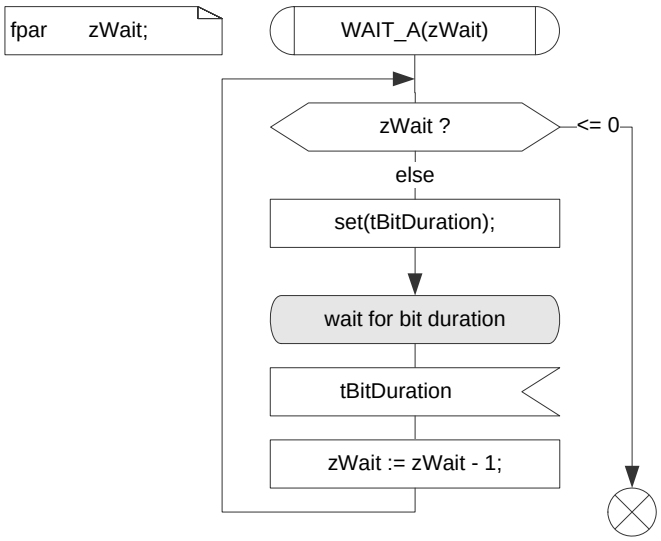

Figure 3-22: Procedure WAIT_A [CODEC_A].

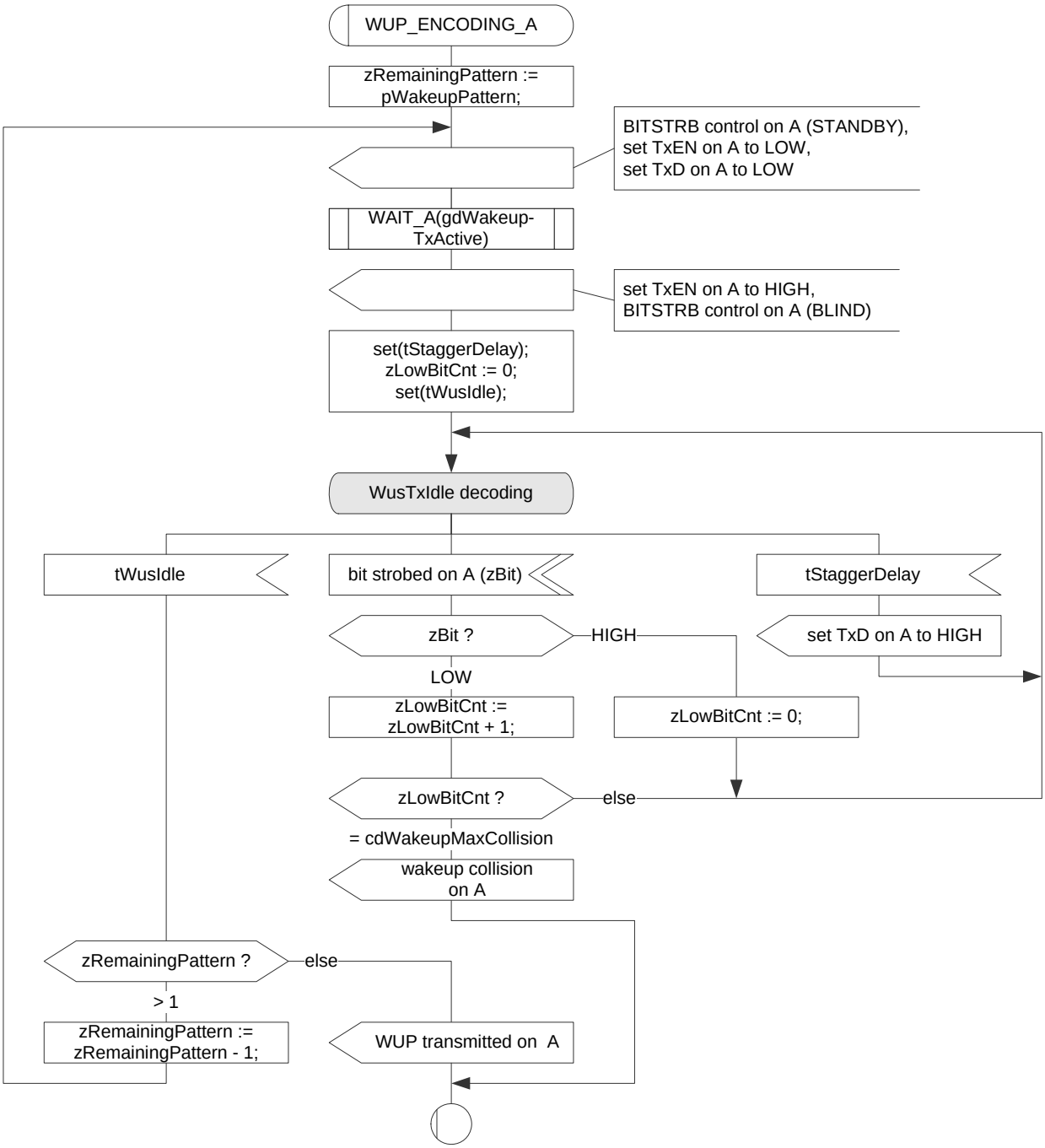

Figure 3-23: Encoding macro WUP_ENCODING_A [CODEC_A].

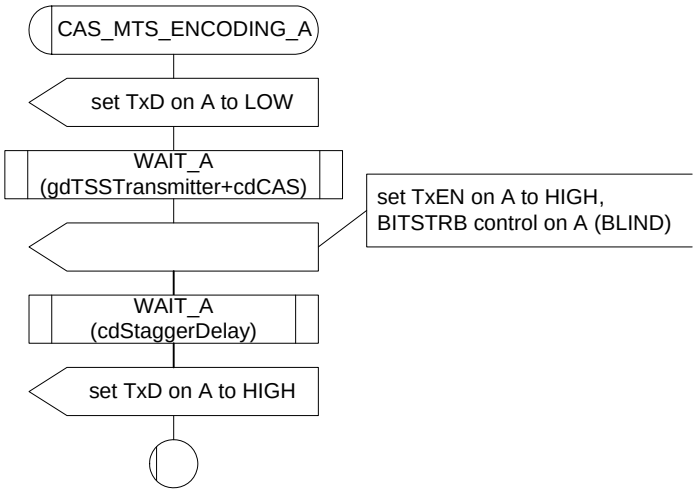

Figure 3-24: Encoding macro CAS_MTS_ENCODING_A [CODEC_A].

Figure 3-25: Encoding macro WUDOP_ENCODING_A [CODEC_A].

3.3.5 Decoding behavior

A receiving node shall decode the received bit stream provided by the BITSTRB process according to the CODEC process.

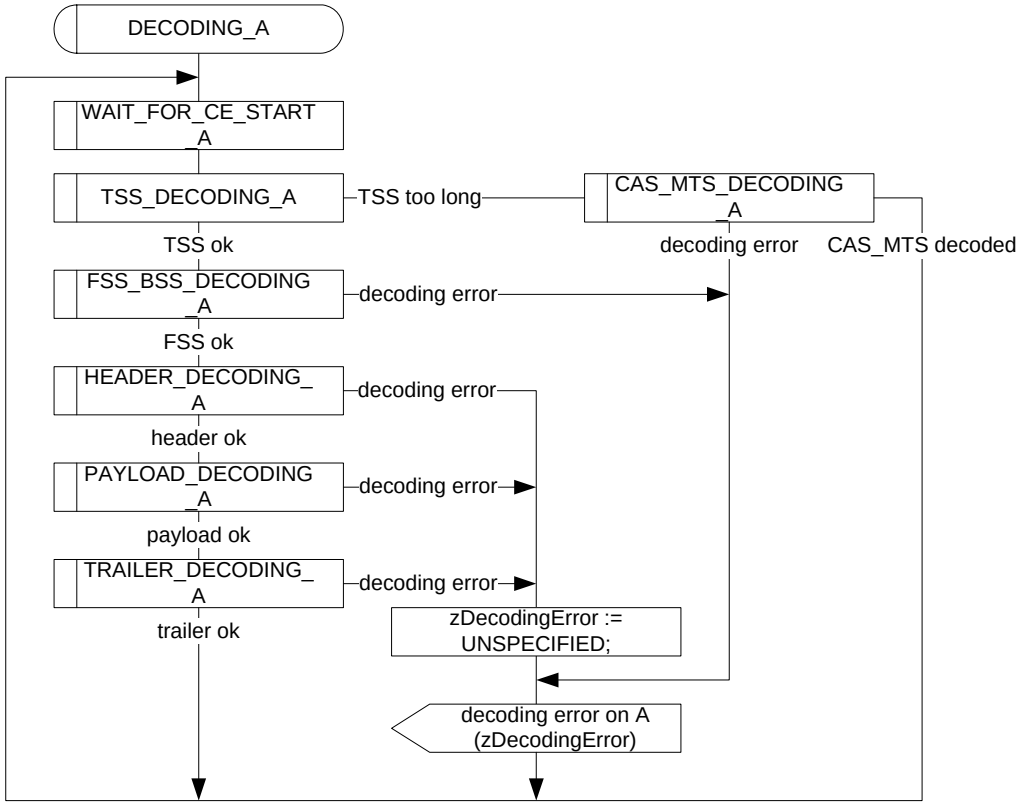

Figure 3-26: Decoding macro DECODING_A [CODEC_A].

3.3.6 Decoding macros

The following formal definitions are used within the frame decoding macros:

```
newtype T_ByteArray
      Array (Integer, T_BitLevel);
endnewtype;
```

Definition 3-6: Formal definition of T_ByteArray.

```
newtype T_ByteStreamArray
      Array (Integer, T_ByteArray);
endnewtype;
```

Definition 3-7: Formal definition of T_ByteStreamArray.

```
syntype
      T_CRCCheckPassed = Boolean
endsyntype;
```

Definition 3-8: Formal definition of T_CRCCheckPassed.

```
syntype
      T_MicrotickTime = Integer
endsyntype;
```

Definition 3-9: Formal definition of T_MicrotickTime.

```
newtype T_ReceiveFrame
struct
      PrimaryTRP  T_MicrotickTime;
      Channel     T_Channel;
      Header      T_Header;
      Payload     T_Payload;
endnewtype;
```

Definition 3-10: Formal definition of T_ReceiveFrame.

```
newtype T_DecodingError
      literals CAS_MTS_TOO_SHORT, FSS_TOO_LONG, UNSPECIFIED;
endnewtype;
```

Definition 3-11: Formal definition of T_DecodingError.

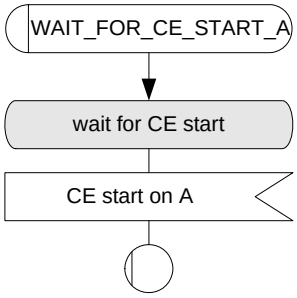

Figure 3-27: Decoding macro WAIT_FOR_CE_START_A [CODEC_A].

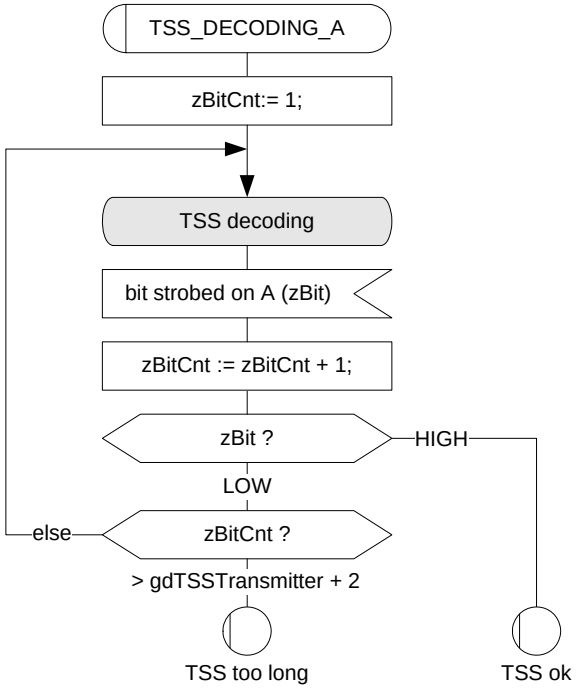

Figure 3-28: Decoding macro TSS_DECODING_A [CODEC_A].

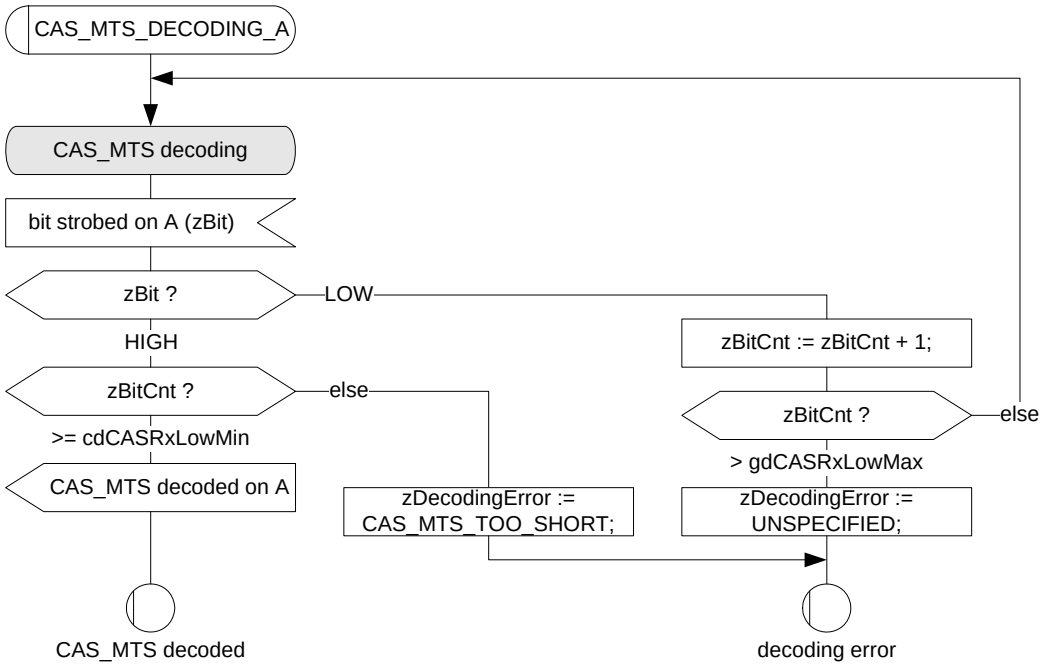

Figure 3-29: Decoding macro CAS_MTS_DECODING_A [CODEC_A].

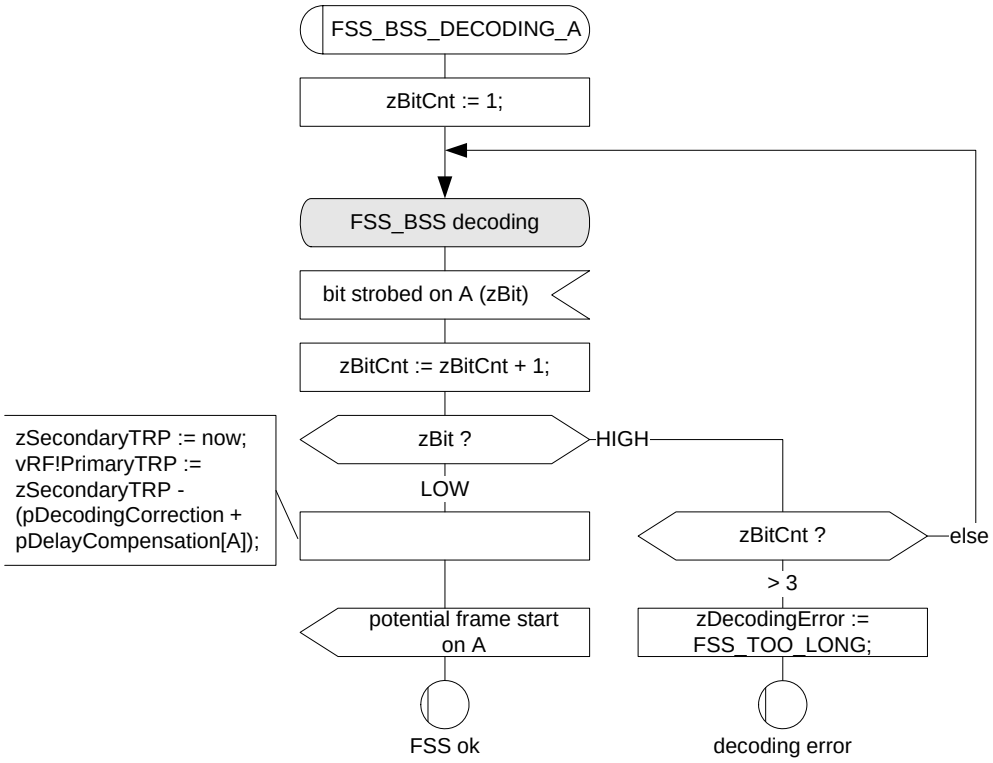

Figure 3-30: Decoding macro FSS_BSS_DECODING_A [CODEC_A].[47]

[47] If a bit is strobed at a microtick boundary 'now' should reflect the larger microtick value.

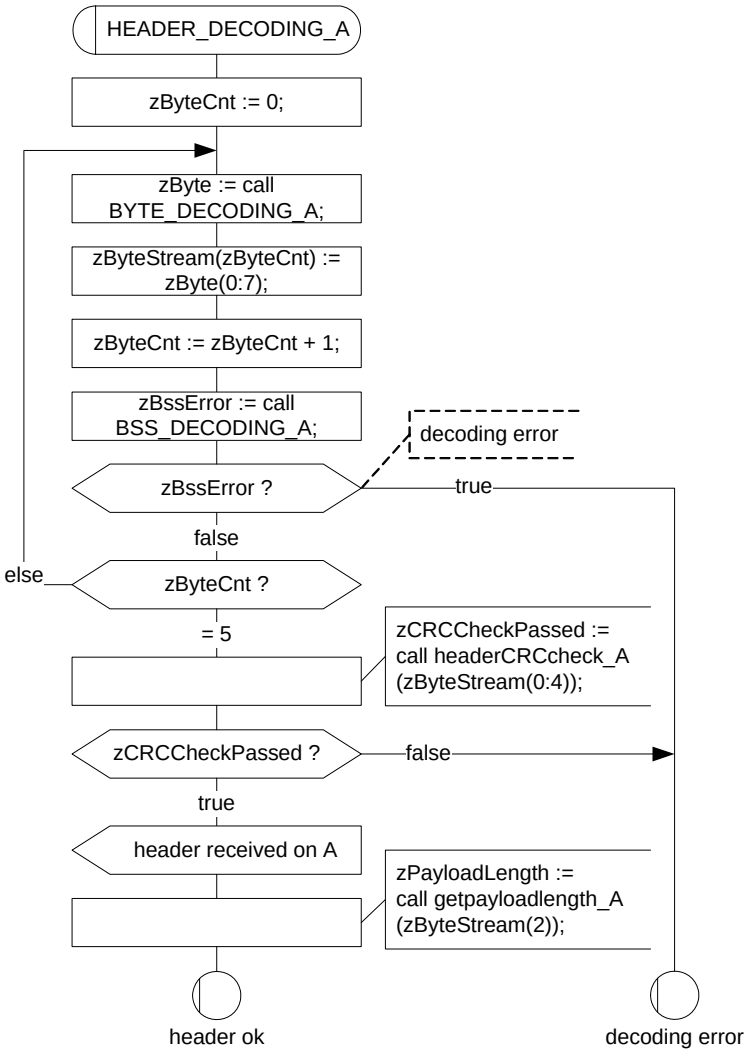

Figure 3-31: Decoding macro HEADER_DECODING_A [CODEC_A].

The function **headerCRCcheck** returns a Boolean, *zCRCCheckPassed*, which is true if the header CRC check was passed (see section 4.5.2) and is false if the header CRC check fails. The function **getpayload-length** returns *zPayloadLength*, the number of bytes in the payload segment of a frame. The CODEC process uses *zPayloadLength* to determine the correct length of a received frame. See also Figure 3-35 and Figure 3-36.

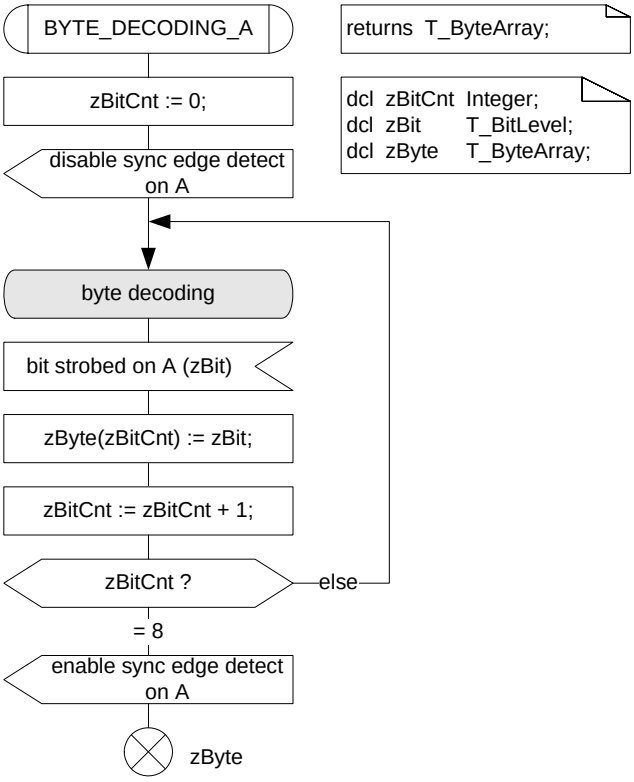

Figure 3-32: Procedure BYTE_DECODING_A [CODEC_A].

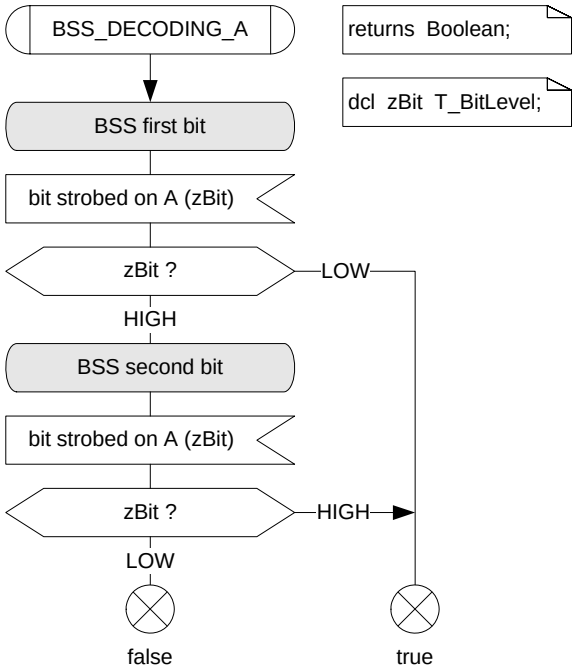

Figure 3-33: Procedure BSS_DECODING_A [CODEC_A].

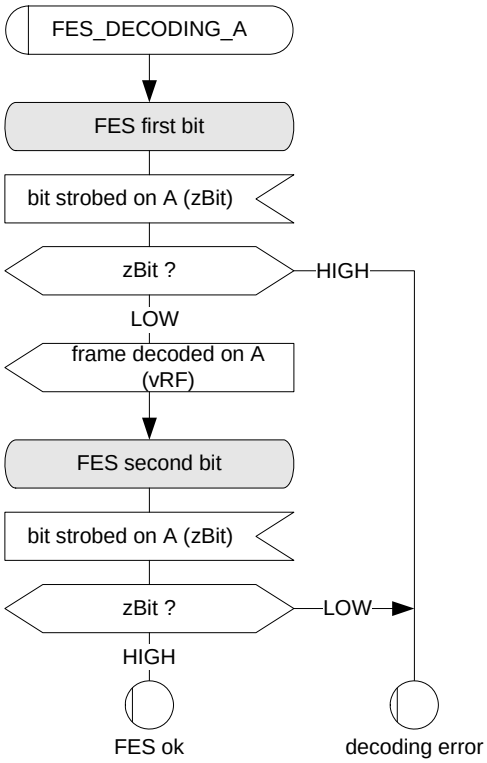

Figure 3-34: Decoding macro FES_DECODING_A [CODEC_A].

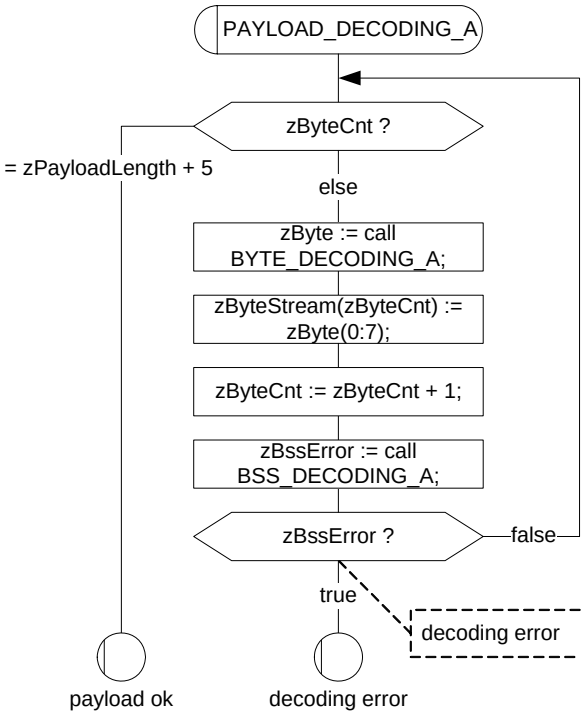

Figure 3-35: Decoding macro PAYLOAD_DECODING_A [CODEC_A].

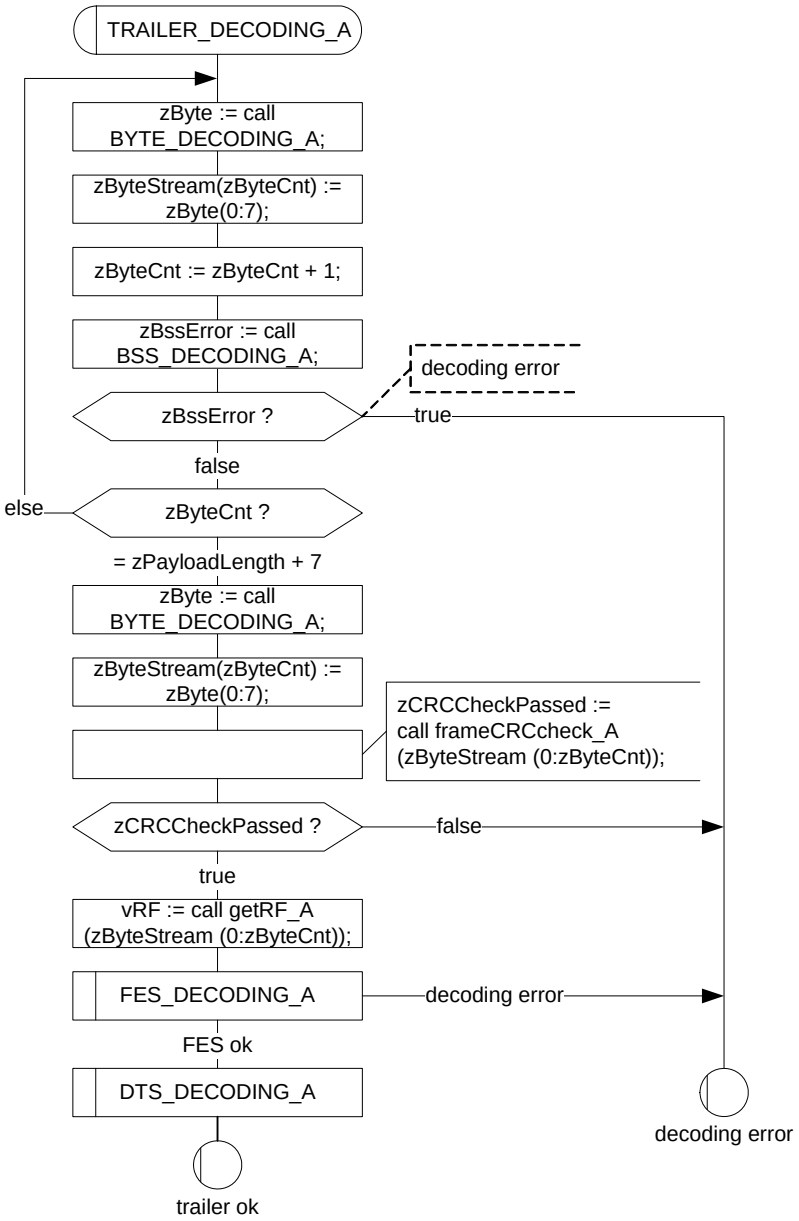

Figure 3-36: Decoding macro TRAILER_DECODING_A [CODEC_A].

The function **frameCRCcheck** returns a Boolean, *zCRCCheckPassed*, which is true if the frame CRC check was passed (see section 4.5.3) and is false if the frame CRC check fails. This function is channel specific due to the channel specific initialization vectors of the CRC calculation (see section 4.5.3 for details).

The function **getRF** used in Figure 3-36 extracts decoded header and payload data from *zByteStream* and returns it via the structure variable *vRF*.

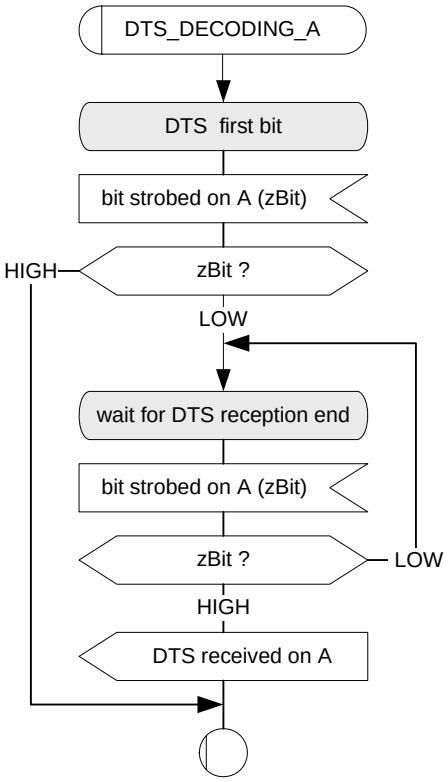

Figure 3-37: Decoding macro DTS_DECODING_A [CODEC_A].

3.4 Bit strobing process

3.4.1 Operating modes

The receiving node shall strobe the received data from the BD according to the bit strobing process BITSTRB. Definition 3-12 shows the formal definition of the BITSTRB operating modes:

```
newtype T_StrbMode
      literals STANDBY, GO, BLIND;
endnewtype;
```

Definition 3-12: Formal definition of T_StrbMode.

The bit strobing process BITSTRB has the following three operating modes:

1. In the STANDBY mode bit strobing is effectively halted.
2. In the GO mode the bit strobing process shall be executed.
3. In the BLIND mode the bit strobing process is paused for a configurable interval. After expiration of this interval the mode is switched automatically to GO.

With the instantiation of the CODEC process the bit strobing process BITSTRB is set in the mode GO and at any transition of the CODEC process to *CODEC:standby* the bit strobing process BITSTRB is set to STANDBY.

3.4.2 Bit strobing process behavior

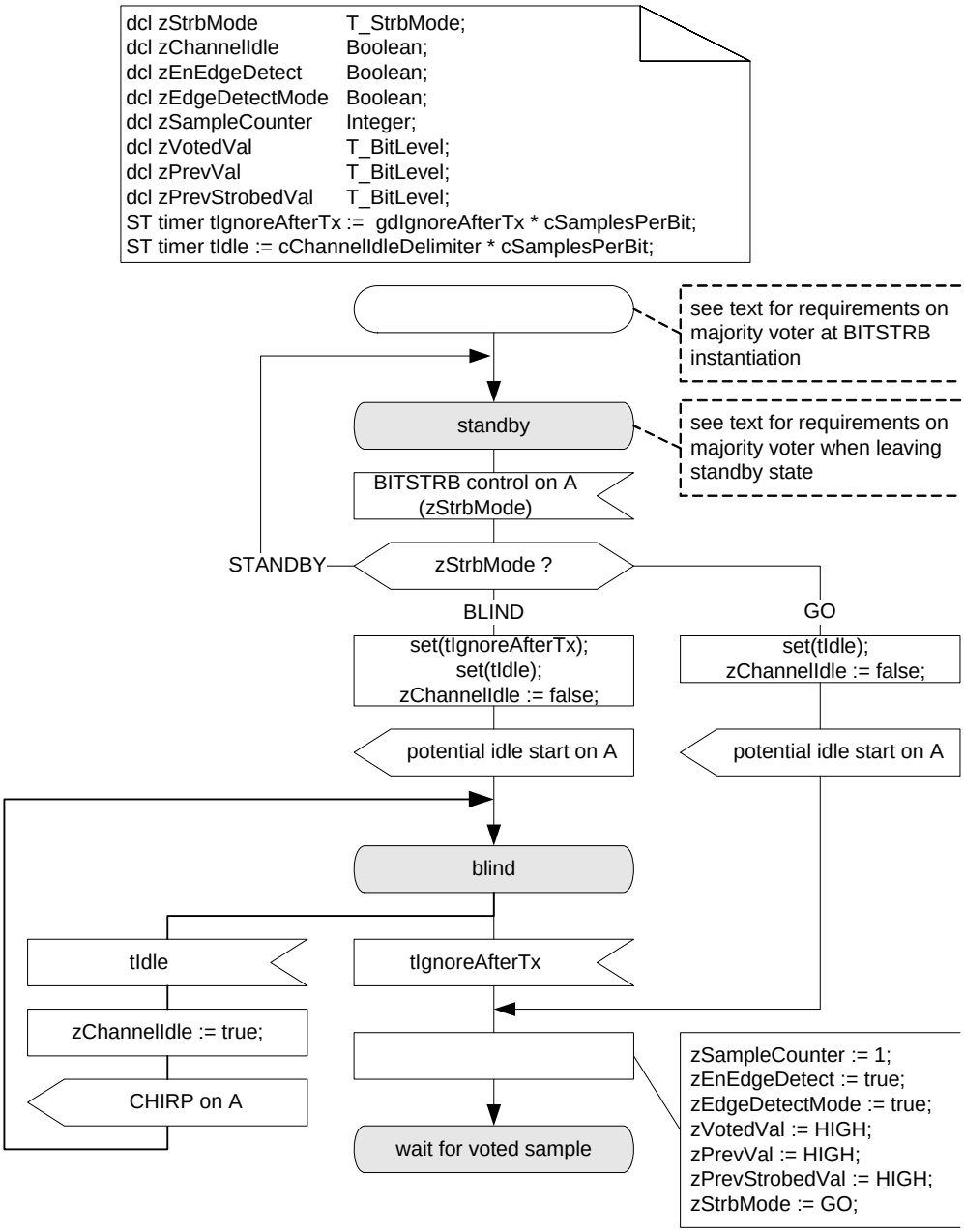

Figure 3-38: BITSTRB process [BITSTRB_A].

The BITSTRB process shall not be instantiated until all stored samples in the majority voter represent the *cVotingSamples* most recent samples of the RxD input, i.e., when initially instantiated the BITSTRB process will immediately begin operation based on real inputs (as opposed to initialization values of the majority voter).

On a transition out of the *BITSTRB:standby* state the majority voter shall behave as if it had been continuously sampling during the entire standby state.[48]

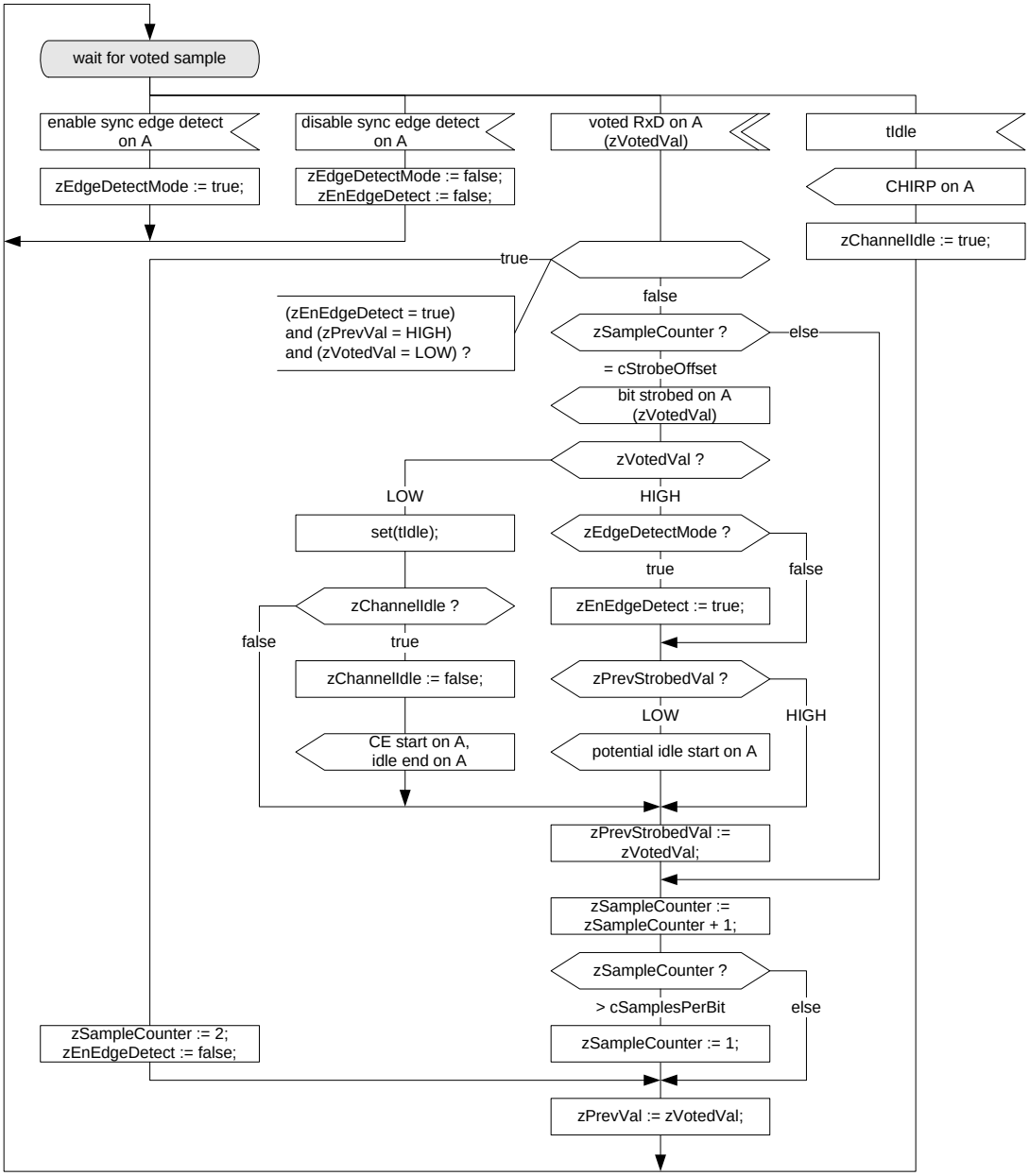

Figure 3-39: Wait for voted sample [BITSTRB_A].

[48] Note that this is not a requirement that an implementation must actually sample when BITSTRB is in the *BITSTRB:standby* state - an implementation that restarted sampling several samples before the transition out of standby would also be acceptable.

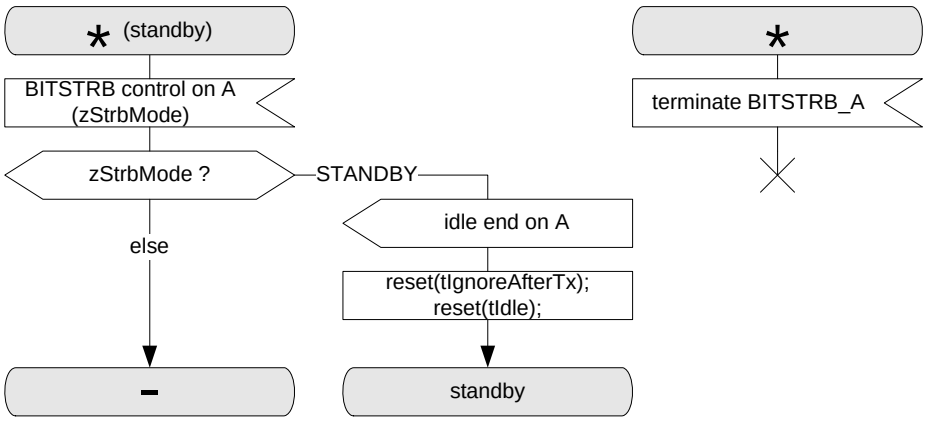

Figure 3-40: BITSTRB process control and process termination [BITSTRB_A].

3.5 Wakeup pattern decoding process

3.5.1 Operating modes

The WUPDEC process is responsible for detecting wakeups present on the bus. Such wakeups could either come from WUPs that are transmitted as part of the initial FlexRay wakeup process or from WUDOPs that are transmitted as part of the wakeup during operation procedure.

The wakeup pattern decoding process WUPDEC distinguishes between two modes. Definition 3-13 shows the formal definition of the WUPDEC operating modes:

```
newtype T_WupDecMode
      literals STANDBY, GO;
endnewtype;
```

Definition 3-13: Formal definition of T_WupDecMode.

1. In the STANDBY mode, the wakeup pattern decoding process (WUPDEC) is effectively halted.
2. In the GO mode, the node shall decode wakeup patterns.

3.5.2 Wakeup pattern decoding process behavior

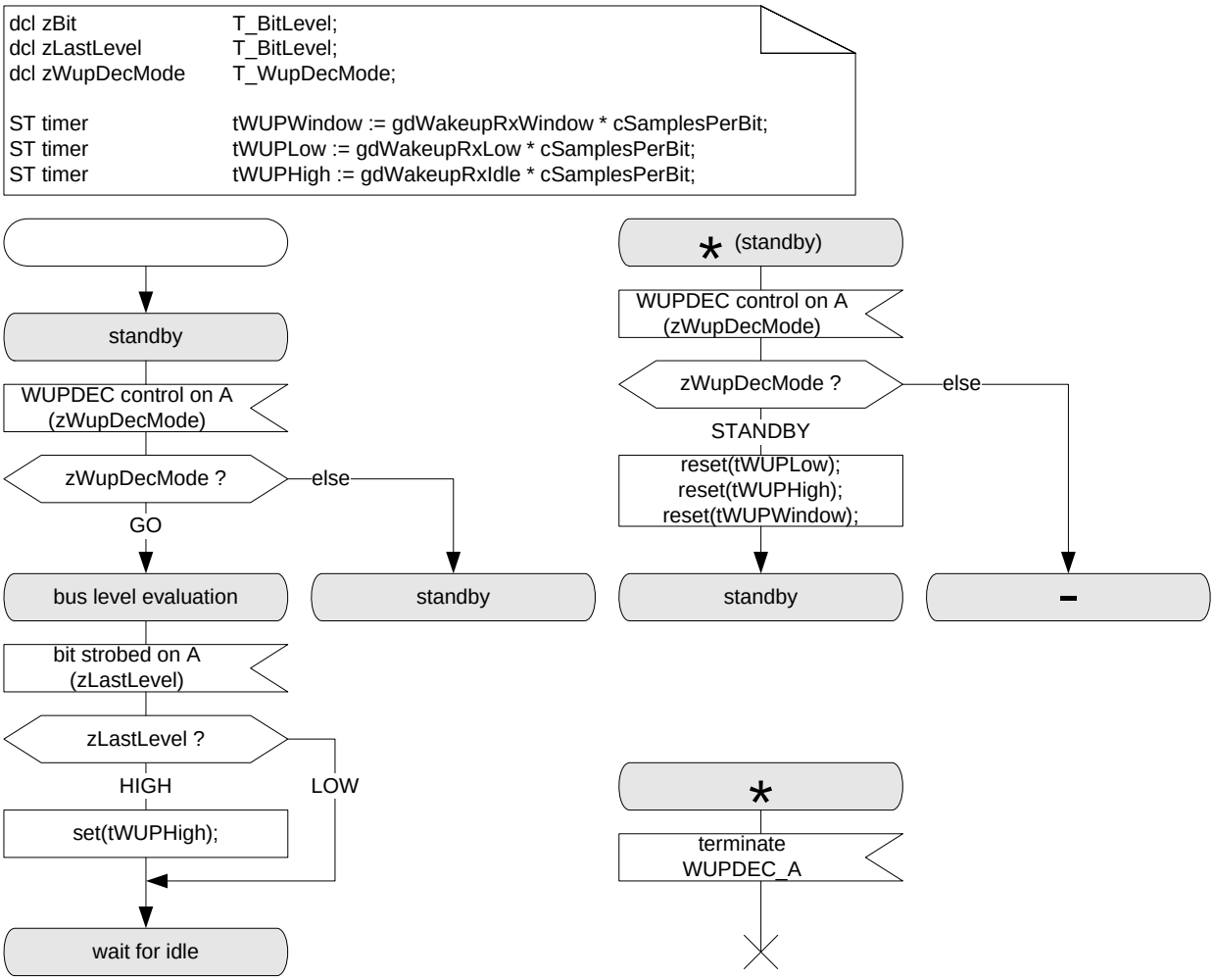

Figure 3-41: Control of the wakeup pattern decoding process and its termination [WUPDEC_A].

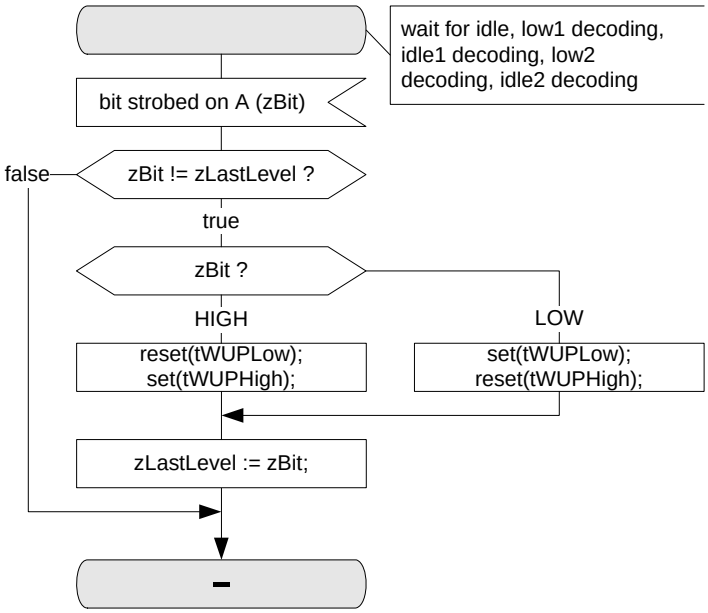

Figure 3-42: Control of the wakeup timer [WUPDEC_A].

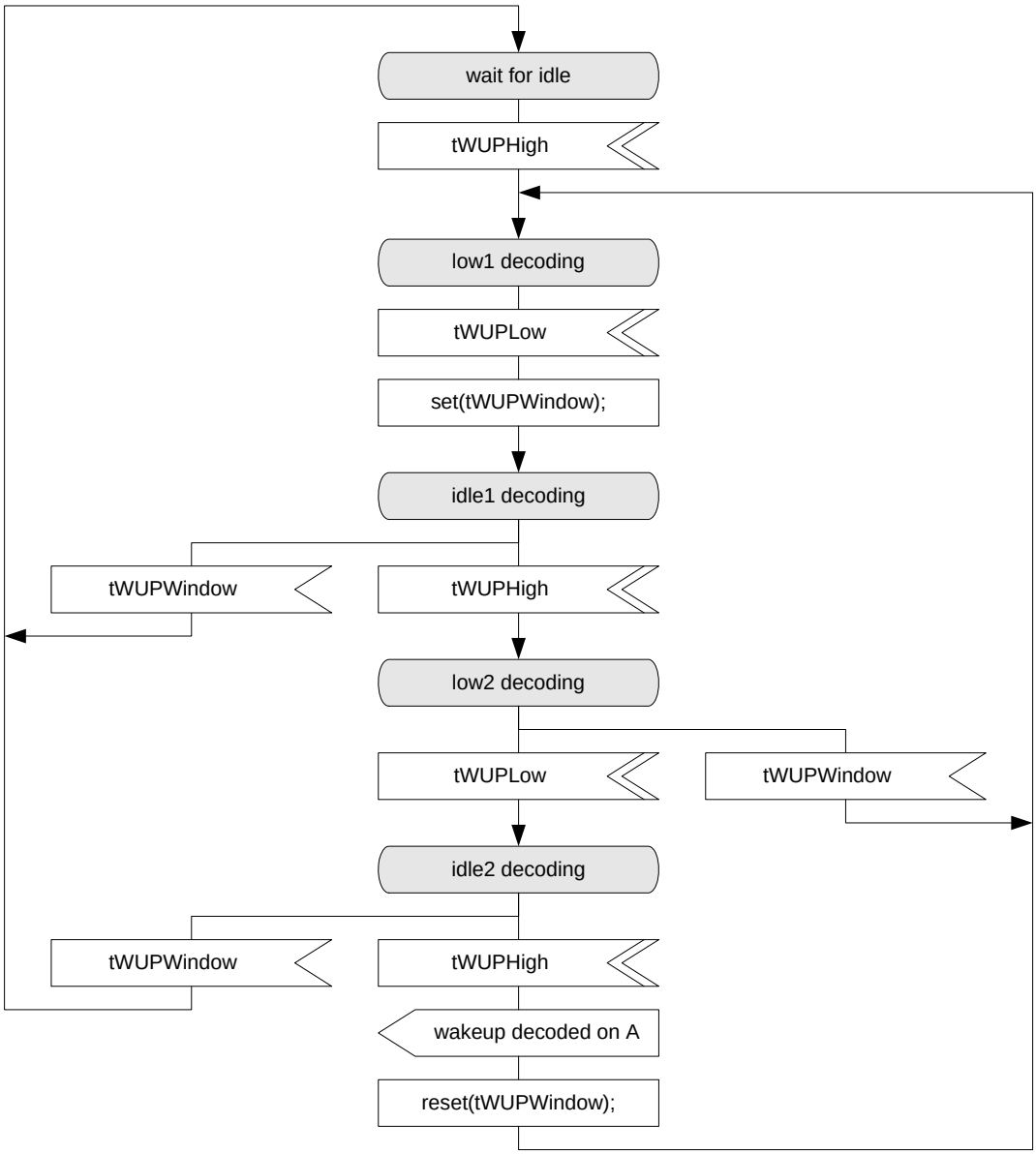

Figure 3-43: Wakeup pattern decoding [WUPDEC_A].

Chapter 4
Frame Format

4.1 Overview

An overview of the FlexRay frame format is given in Figure 4-1. The frame consists of three segments. These are the header segment, the payload segment, and the trailer segment.

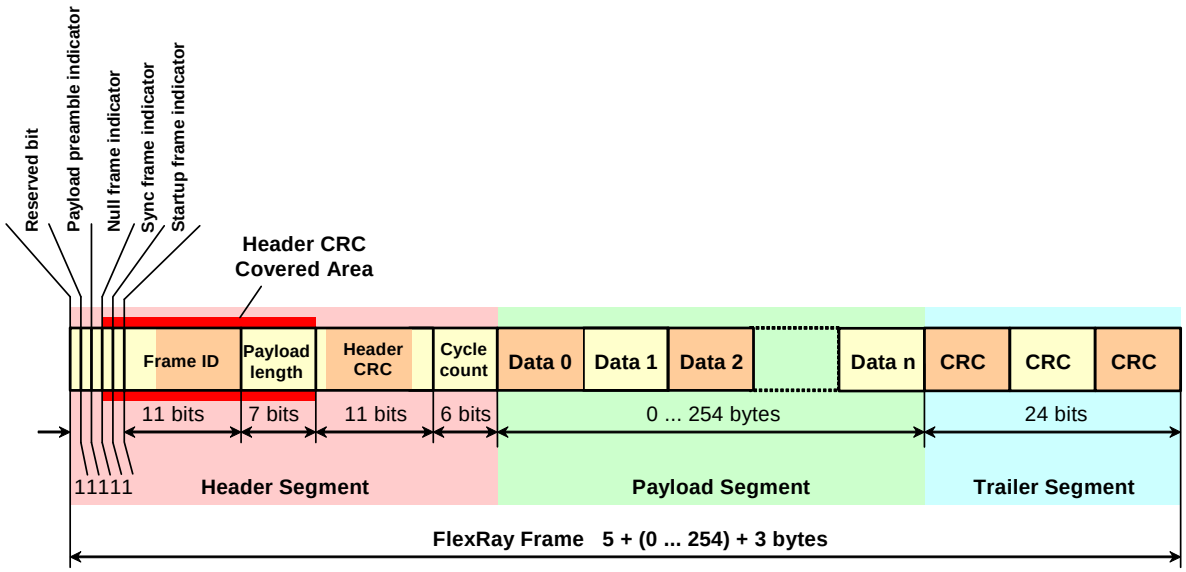

Figure 4-1: FlexRay frame format.

The node shall transmit the frame on the network such that the header segment appears first, followed by the payload segment, and then followed by the trailer segment, which is transmitted last. Within the individual segments the node shall transmit the fields in left to right order as depicted in Figure 4-1, (in the header segment, for example, the reserved bit is transmitted first and the cycle count field is transmitted last).

4.2 FlexRay header segment (5 bytes)

The FlexRay header segment consists of 5 bytes. These bytes contain the reserved bit, the payload preamble indicator, the null frame indicator, the sync frame indicator, the startup frame indicator, the frame ID, the payload length, the header CRC, and the cycle count.

4.2.1 Reserved bit (1 bit)

The reserved bit is reserved for future protocol use. It shall not be used by the application.

- A transmitting node shall set the reserved bit to logical '0'.
- A receiving node shall ignore the reserved bit.[49]

```
syntype T_Reserved = Integer
      constants 0 : 1
endsyntype;
```

Definition 4-1: Formal definition of T_Reserved.

4.2.2 Payload preamble indicator (1 bit)

The payload preamble indicator indicates whether or not an optional vector is contained within the payload segment of the frame transmitted:

- If the frame is transmitted in the static segment the payload preamble indicator indicates the presence of a network management vector at the beginning of the payload.
- If the frame is transmitted in the dynamic segment the payload preamble indicator indicates the presence of a message ID at the beginning of the payload.

If the payload preamble indicator is set to zero then the payload segment of the frame does not contain a network management vector or a message ID, respectively.

If the payload preamble indicator is set to one then the payload segment of the frame contains a network management vector if it is transmitted in the static segment or a message ID if it is transmitted in the dynamic segment.

```
syntype T_PPIndicator = Integer
      constants 0 : 1
endsyntype;
```

Definition 4-2: Formal definition of T_PPIndicator.

4.2.3 Null frame indicator (1 bit)

The null frame indicator indicates whether or not the frame is a null frame, i.e. a frame that contains no useable data in the payload segment of the frame.[50] The conditions under which a transmitting node may send a null frame are detailed in Chapter 5. Nodes that receive a null frame may still use some information related to the frame.[51]

- If the null frame indicator is set to zero then the payload segment contains no valid data. All bytes in the payload segment are set to zero, and the payload preamble indicator is set to zero.
- If the null frame indicator is set to one then the payload segment contains data.

```
syntype T_NFIndicator = Integer
      constants 0 : 1
endsyntype;
```

Definition 4-3: Formal definition of T_NFIndicator.

4.2.4 Sync frame indicator (1 bit)

The sync frame indicator indicates whether or not the frame is a sync frame, i.e. a frame that is utilized for system wide synchronization of communication.[52]

---

[49] The receiving node uses the value of the reserved bit for the Frame CRC checking process, but otherwise ignores its value (i.e., the receiver shall accept either a 1 or a 0 in this field).

[50] The null frame indicator indicates only whether payload data was available to the communication controller at the time the frame was sent. A null frame indicator set to zero informs the receiving node(s) that data in the payload segment must not be used. If the bit is set to one data is present in the payload segment from the transmitting communication controller's perspective. The receiving node may still have to do additional checks to decide whether the data is actually valid from an application perspective.

[51] For example, the clock synchronization algorithm will use the arrival time of null frames with the Sync frame indicator set to one (provided all other criteria for that frame's acceptance are met).

[52] Sync frames are only sent in the static segment. Please refer to the rules to configure sync frames.

- If the sync frame indicator is set to zero then no receiving node shall consider the frame for synchronization or synchronization related tasks.
- If the sync frame indicator is set to one then all receiving nodes shall utilize the frame for synchronization if it meets other acceptance criteria (see below).

The clock synchronization mechanism (described in Chapter 8) makes use of the sync frame indicator. There are several conditions that result in the sync frame indicator being set to one and subsequently utilized by the clock synchronization mechanism. Details of how the node shall set the sync frame indicator are specified in Chapter 5 and section 8.8.

```
syntype T_SyFIndicator = Integer
      constants 0 : 1
endsyntype;
```

Definition 4-4: Formal definition of T_SyFIndicator.

4.2.5 Startup frame indicator (1 bit)

The startup frame indicator indicates whether or not a frame is a startup frame. Startup frames serve a special role in the startup mechanism. Only coldstart nodes are allowed to transmit startup frames.

- If the startup frame indicator is set to zero then the frame is not a startup frame.
- If the startup frame indicator is set to one then the frame is a startup frame.

The startup frame indicator shall only be set to one in the sync frames of coldstart nodes. Therefore, a frame with the startup frame indicator set to one shall also have the sync frame indicator set to one. As a result, all valid startup frames are also sync frames.

The startup (described in Chapter 7) and clock synchronization (described in Chapter 8) mechanisms utilize the startup frame indicator. In both cases, the condition that the startup frame indicator is set to one is only one of several conditions necessary for the frame to be used by the mechanisms. Details regarding how the node sets the startup frame indicator are specified in Chapter 5.[53]

```
syntype T_SuFIndicator = Integer
      constants 0 : 1
endsyntype;
```

Definition 4-5: Formal definition of T_SuFIndicator.

4.2.6 Frame ID (11 bits)

The frame ID defines the slot in which the frame should be transmitted. A frame ID is used no more than once on each channel in a communication cycle. Each frame that may be transmitted in a cluster has a frame ID assigned to it.

The frame ID ranges from 1 to 2047[54]. The frame ID 0 is an invalid frame ID[55].

The node shall transmit the frame ID such that the most significant bit of the frame ID is transmitted first with the remaining bits of the frame ID being transmitted in decreasing order of significance.

```
syntype T_FrameID = Integer
      constants 0 : 2047[56]
```

[53] The configuration of exactly three nodes in a cluster as coldstart nodes avoids the formation of cliques during startup for several fault scenarios. It is also possible to configure more than three nodes as coldstart nodes but the clique avoidance mechanism will not work in this case.

[54] In binary: from $(000\ 0000\ 0001)_2$ to $(111\ 1111\ 1111)_2$

[55] The frame ID of a transmitted frame is determined by the value of *vSlotCounter(Ch)* at the time of transmission (see Chapter 5). In the absence of faults, *vSlotCounter(Ch)* can never be zero when a slot is available for transmission. Received frames with frame ID zero will always be identified as erroneous because a slot ID mismatch is a certainty due to the fact that there is no slot with ID zero.

[56] Frame IDs range from 1 to 2047. The zero is used to mark invalid frames, empty slots, etc.

```
endsyntype;
```

Definition 4-6: Formal definition of T_FrameID.

4.2.7 Payload length (7 bits)

The payload length field is used to indicate the size of the payload segment. The payload segment size is encoded in this field by setting it to the number of payload data bytes divided by two (e.g., a frame that contains a payload segment consisting of 72 bytes would be sent with the payload length set to 36).[57]

The payload length ranges from 0 to *cPayloadLengthMax*[58], which corresponds to a payload segment containing 2 * *cPayloadLengthMax* bytes.

The payload length shall be fixed and identical for all frames sent in the static segment of a communication cycle. For these frames the payload length field shall be transmitted with the payload length set to *gPayloadLengthStatic*.

The payload length may be different for different frames in the dynamic segment of a communication cycle. In addition, the payload length of a specific dynamic segment frame may vary from cycle to cycle. Finally, the payload lengths of a specific dynamic segment frame may be different on each configured channel.

The node shall transmit the payload length such that the most significant bit of the payload length is transmitted first with the remaining bits of the payload length being transmitted in decreasing order of significance.

```
syntype T_Length = Integer
      constants 0 : cPayloadLengthMax
endsyntype;
```

Definition 4-7: Formal definition of T_Length.

4.2.8 Header CRC (11 bits)

The header CRC contains a cyclic redundancy check code (CRC) that is computed over the sync frame indicator, the startup frame indicator, the frame ID, and the payload length. The CC shall not calculate the header CRC for a transmitted frame. The header CRC of transmitted frames is computed offline and provided to the CC by means of configuration (i.e., it is not computed by the transmitting CC).[59] The CC shall calculate the header CRC of a received frame in order to check that the CRC is correct.

The CRC is computed in the same manner for all configured channels. The CRC polynomial[60] shall be

$$x^{11} + x^9 + x^8 + x^7 + x^2 + 1 = (x+1) \cdot (x^5 + x^3 + 1) \cdot (x^5 + x^4 + x^3 + x + 1)$$

The initialization vector of the register used to generate the header CRC shall be 0x01A.

With respect to the computation of the header CRC, the sync frame indicator shall be shifted in first, followed by the startup frame indicator, followed by the most significant bit of the frame ID, followed by subsequent bits of the frame ID, followed by the most significant bit of the payload length, and followed by subsequent bits of the payload length.

[57] The payload length field does not include the number of bytes within the header and the trailer segments of the FlexRay frame.

[58] The electrical physical layer puts a restriction on the usable payload such that the overall transmission duration is limited (see [EPL10]. It may restrict the payload length at 2.5 and 5 Mbit/s. For details please refer to sections B.4.40 and B.4.41.

[59] For a given frame in the static segment the values of the header fields covered by the CRC do not change during the operation of the cluster in the absence of faults. Implicitly, the CRC does not need to change either. Offline calculation of the CRC makes it unlikely that a fault-induced change to the covered header fields will also result in a frame with a valid header CRC (since the CRC is not recalculated based on the modified header fields).

[60] This 11 bit CRC polynomial generates a (31,20) BCH code that has a minimum Hamming distance of 6. The codeword consists of the data to be protected and the CRC. In this application, this CRC protects exactly 20 bits of data (1 sync frame indicator bit + 1 startup frame indicator bit + 11 frame ID bits + 7 payload length bits = 20 bits). This polynomial was obtained from [Wad01] and its properties were verified using the techniques described in [Koo02].

The node shall transmit the header CRC such that the most significant bit of the header CRC is transmitted first with the remaining bits of the header CRC being transmitted in decreasing order of significance.

A detailed description of how to generate and verify the header CRC is given in section 4.5.2.

```
syntype T_HeaderCRC = Integer
      constants 0 : 2047
endsyntype;
```

Definition 4-8: Formal definition of T_HeaderCRC.

4.2.9 Cycle count (6 bits)

The cycle count indicates the transmitting node's view of the value of the cycle counter *vCycleCounter* at the time of frame transmission (see section 5.3.2.2 and section 5.3.3.2).

The node shall transmit the cycle count such that the most significant bit of the cycle count is transmitted first with the remaining bits of the cycle count being transmitted in decreasing order of significance.

```
syntype T_CycleCounter = Integer
      constants 0 : 63
endsyntype;
```

Definition 4-9: Formal definition of T_CycleCounter.

4.2.10 Formal header definition

The formal definitions of the fields in the previous sections and the header segment structure depicted in Figure 4-1 yield the following formal definition for the header segment:

```
newtype T_Header
struct
      Reserved          T_Reserved;
      PPIndicator       T_PPIndicator;
      NFIndicator       T_NFIndicator;
      SyFIndicator      T_SyFIndicator;
      SuFIndicator      T_SuFIndicator;
      FrameID           T_FrameID;
      Length            T_Length;
      HeaderCRC         T_HeaderCRC;
      CycleCount        T_CycleCounter;
endnewtype;
```

Definition 4-10: Formal definition of T_Header.

4.3 FlexRay payload segment (0 - 254 bytes)

The FlexRay payload segment contains 0 to 254[61] bytes (0 to 127 two-byte words) of data. Because the payload length contains the number of two-byte words, the payload segment contains an even number of bytes. The bytes of the payload segment are identified numerically, starting at 0 for the first byte after the header segment and increasing by one with each subsequent byte. The individual bytes are referred to as "Data 0", "Data 1", "Data 2", etc., with "Data 0" being the first byte of the payload segment, "Data 1" being the second byte, etc.

The frame CRC described in section 4.5.3 has a Hamming distance of six for payload lengths up to and including 248 bytes. For payload lengths greater than 248 bytes the CRC has a Hamming distance of four.

[61] The electrical physical layer puts a restriction on the usable payload such that the overall transmission duration is limited (see [EPL10]). It may restrict the payload length at 2.5 and 5 Mbit/s. For details please refer to sections B.4.40 and B.4.41.

For frames transmitted in the dynamic segment the first two bytes of the payload segment may optionally be used as a message ID field, allowing receiving nodes to filter or steer data based on the contents of this field. The payload preamble indicator in the frame header indicates whether the payload segment contains the message ID.

For frames transmitted in the static segment the first 0 to 12 bytes of the payload segment may be used as a network management vector. The payload preamble indicator in the frame header indicates whether the payload segment contains the network management vector[62]. The length of the network management vector *gNetworkManagementVectorLength* is configured during the *POC:config* state and cannot be changed in any other state. *gNetworkManagementVectorLength* can be configured between 0 and 12 bytes, inclusive.

Starting with payload "Data 0" the node shall transmit the bytes of the payload segment such that the most significant bit of the byte is transmitted first with the remaining bits of the byte being transmitted in decreasing order of significance.[63]

The product specific host interface specification determines the mapping between the position of bytes in the buffer and the position of the bytes in the payload segment of the frame.

```
newtype T_Payload
      Array(T_Length, Integer)
endnewtype;
```

Definition 4-11: Formal definition of T_Payload.

4.3.1 NMVector

A number of bytes in the payload segment of the FlexRay frame format in a frame transmitted in the static segment can be used as Network Management Vector (NMVector).

- The length of the NMVector is configured during *POC:config* by the parameter *gNetworkManagementVectorLength*. All nodes in a cluster must be configured with the same value for this parameter.
- The NMVector may only be used in frames transmitted in the static segment.
- At the transmitting node the NMVector is written by the host as application data. The communication controller has no knowledge about the NMVector and no mechanism inside the communication controller is based on the NMVector except the ORing function described in section 9.3.3.4.
- The optional presence of NMVector is indicated in the frame header by the payload preamble indicator.
- The bits in a byte of the NMVector shall be transmitted such that the most significant bit of a byte is transmitted first followed by the remaining bits being transmitted in decreasing order of significance.
- The least significant byte of the NMVector is transmitted first followed by the remaining bytes in increasing order of significance.[64]

[62] Frames that contain network management data are not restricted to contain only network management data - the other bytes in the payload segment may be used to convey additional, non-Network Management data.

[63] If a message ID exists, the most significant byte of the message ID is transmitted first followed by the least significant byte of the message ID. If no message ID exists the transmission starts with the first payload data byte (Data 0) followed by the remaining payload data bytes.

[64] This allows lower bits to remain at defined positions if the length of the NMVector changes.

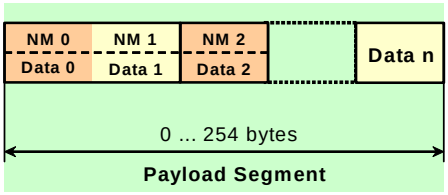

Figure 4-2: Payload segment of frames transmitted in the static segment.

4.3.2 Message ID (16 bits)

The first two bytes of the payload segment of the FlexRay frame format for frames transmitted in the dynamic segment can be used as receiver filterable data called the message ID.

- The message ID is an application determined number that identifies the contents of the data segment.
- The message ID can only be used in frames transmitted in the dynamic segment.
- The message ID is 16 bits long.
- At the transmitting node the message ID is written by the host as application data. The protocol engine has no knowledge about the message ID and no mechanism inside the protocol engine is based on the message ID.
- At the receiving node the storage of a frame may depend on the result of a filtering process that makes use of the message ID. All frame checks done in Frame Processing (see Chapter 6) are unmodified (i.e., are not a function of the message ID). The use of the message ID filter is defined in section 9.3.2.10.2.5.
- The presence or absence of a message ID is indicated in the frame header by the payload preamble indicator.
- If this mechanism is used, the most significant bit of the MessageID shall be placed in the most significant bit of the first byte of the payload segment. Subsequent bits of the message ID shall be placed in the next payload bits in order of decreasing significance.

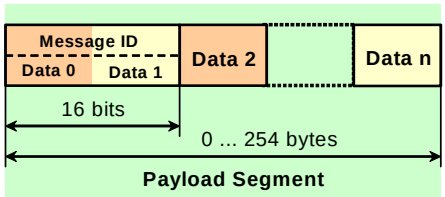

Figure 4-3: Payload segment of frames transmitted in the dynamic segment.

4.4 FlexRay trailer segment

The FlexRay trailer segment contains a single field, a 24-bit CRC for the frame.

The Frame CRC field contains a cyclic redundancy check code (CRC) computed over the header segment and the payload segment of the frame. The computation includes all fields in these segments.[65]

The CRC is computed using the same generator polynomial on both channels. The CRC polynomial[66] shall be

[65] This includes the header CRC, as well as any Communication Controller-generated "padding" bytes that may be included in the payload segment.

$$x^{24} + x^{22} + x^{20} + x^{19} + x^{18} + x^{16} + x^{14} + x^{13} + x^{11} + x^{10} + x^8 + x^7 + x^6 + x^3 + x + 1$$
$$= (x+1)^2 \cdot (x^{11} + x^9 + x^8 + x^7 + x^5 + x^3 + x^2 + x + 1) \cdot (x^{11} + x^9 + x^8 + x^7 + x^6 + x^3 + 1)$$

The node shall use a different initialization vector depending on which channel the frame should be transmitted[67]:

- The node shall use the initialization vector `0xFEDCBA` for frames sent on channel A.
- The node shall use the initialization vector `0xABCDEF` for frames sent on channel B.

With respect to the computation of the frame CRC, the frame fields shall be fed into the CRC generator in network order starting with the reserved bit, and ending with the least significant bit of the last byte of the payload segment.

The frame CRC shall be transmitted such that the most significant bit of the frame CRC is transmitted first with the remaining bits of the frame CRC being transmitted in decreasing order of significance.

A detailed description of how to generate or verify the Frame CRC is given in section 4.5.3.

```
syntype T_FrameCRC = Integer
      constants 0x000000 : 0xFFFFFF
endsyntype;
```

Definition 4-12: Formal definition of T_FrameCRC.

4.5 CRC calculation details

The behavior of the CODEC while processing a received frame depends on whether or not the received header and frame CRCs are verified to match the values locally calculated using the actual frame data (see Figure 3-31 and Figure 3-36). The CODEC also appends the CRC in the trailer segment of a transmitted frame (see section 3.2.1.1.6). The algorithm executed to calculate the CRC is the same in all cases except for the initial values of several algorithm parameters (see below).

4.5.1 CRC calculation algorithm

Initialize the CRC shift register with the appropriate initialization value. As long as bits (vNextBit) from the header or payload segment of the frame are available the while-loop is executed. The number of bits available in the payload segment is derived from the payload length field. The bits[68] of the header and payload segments are fed into the CRC register by using the variable vNextBit, bit by bit, in network order, e.g., for the FlexRay frame CRC calculation the first bit used as vNextBit is the reserved bit field, and the last bit used is the least significant bit of the last byte of the payload segment.

The following pseudo code summarizes the CRC calculation algorithm:

```
vCrcReg(vCrcSize - 1 : 0) = vCrcInit;     // Initialize the CRC register
while(vNextBit)
```

[66] This 24-bit CRC polynomial generates a code that has a minimum Hamming distance of 6 for codewords up to 2048 bits in length and a minimum Hamming distance of 4 for codewords up to 4094 bits in length. The codeword consists of all frame data and the CRC. This corresponds to H=6 protection for FlexRay frames with payload lengths up to 248 bytes and H=4 protection for longer payload lengths. This polynomial was obtained from[Cas93] , and its properties were verified using the techniques described in[Koo02] .

[67] Different initialization vectors are defined to prevent a node from communicating if it has crossed channels, connection of a single channel node to the wrong channel, or shorted channels (both controller channels connected to the same physical channel).

[68] Transmitting nodes use the bit sequence that will be fed into the coding algorithm (see Chapter 3), including any controller generated padding bits. Receivers use the decoded sequence as received from the decoding algorithm (i.e., after the removal of any coding sequences (e.g. Byte Start Sequences, Frame Start Sequences, etc.)).

```
      // determine if the CRC polynomial has to be applied by taking
      // the exclusive OR of the most significant bit of the CRC register
      // and the next bit to be fed into the register

      vCrcNext = vNextBit EXOR vCrcReg(vCrcSize - 1);

      // Shift the CRC register left by one bit

      vCrcReg (vCrcSize - 1 : 1) = vCrcReg(vCrcSize - 2 : 0);
      vCrcReg(0) = 0;

      // Apply the CRC polynomial if necessary

      if vCrcNext
            vCrcReg(vCrcSize - 1 : 0) =
                  vCrcReg(vCrcSize - 1 : 0) EXOR vCrcPolynomial;
      end;                        // end if
end;                              // end while loop
```

4.5.2 Header CRC calculation

Among its other uses, the header CRC field of a FlexRay frame is intended to provide protection against improper modification of the sync frame indicator or startup frame indicator fields by a faulty communication controller (CC). The CC that is responsible for transmitting a particular frame shall **not** compute the header CRC field for that frame. Rather, the CC shall be configured with the appropriate header CRC for a given frame by the host[69].

When a CC receives a frame it shall perform the header CRC computations based on the header field values received and check the computed value against the header CRC value received in the frame. The frames from each channel are processed independently. The algorithm described in section 4.5.1 is used to calculate the header CRC. The parameters for the algorithm are defined as follows:

FlexRay header CRC calculation algorithm parameters:

```
vCrcSize = cHCrcSize;              // (= 11) size of the register is 11 bits
vCrcInit = cHCrcInit;              // (= 0x1A) initialization vector of header
                                   // CRC for both channels
vCrcPolynomial = cHCrcPolynomial;  // (= 0x385) hexadecimal representation of
                                   // the header CRC polynomial
```

The results of the calculation (vCrcReg) are compared to the header CRC value in the frame. If the calculated and received values match the header CRC check passes, otherwise it fails.

4.5.3 Frame CRC calculation

The Frame CRC calculation is done inside the communication controller before transmission or after reception of a frame. It is part of the frame transmission process and the frame reception process.

When a CC receives a frame it shall perform the frame CRC computations based on the header and payload field values received and check the computed value against the frame CRC value received in the frame. The frames from each channel are processed independently. The algorithm described in section 4.5.1 is used to calculate the header CRC. The parameters for the algorithm are defined as follows:

[69] This makes it unlikely that a fault in the CC that causes the value of a sync or startup frame indicator to change would result in a frame that is accepted by other nodes in the network because the header CRC would not match. Removing the capability of the transmitter to generate the CRC minimizes the possibility that a frame that results from a CC fault would have a proper header CRC.

FlexRay frame CRC calculation algorithm parameters - channel A:

```
vCrcSize = cCrcSize;                // (= 24) size of the register is 24 bits
vCrcInit = cCrcInit[A];             // (= 0xFEDCBA) initialization vector of
                                    // channel A
vCrcPolynomial = cCrcPolynomial;    // (= 0x5D6DCB) hexadecimal representation
                                    // of the CRC polynomial
```

FlexRay frame CRC calculation algorithm parameters - channel B:

```
vCrcSize = cCrcSize;                // (= 24) size of the register is 24 bits
vCrcInit = cCrcInit[B];             // (= 0xABCDEF) initialization vector of
                                    // channel B

vCrcPolynomial = cCrcPolynomial;    // (= 0x5D6DCB) hexadecimal representation
                                    // of the CRC polynomial
```

The results of the calculation (vCrcReg) are compared to the frame CRC value in the frame on the appropriate channel. If the calculated and received values match the frame CRC check passes, otherwise it fails.

The frame CRC value used in the trailer segment of a transmitted frame is calculated using the same algorithm and the same algorithm parameters, but it is calculated using the data content of the frame to be transmitted.

Chapter 5
Media Access Control

This chapter defines how the node shall perform media access control.

5.1 Principles

In the FlexRay protocol, media access control is based on a recurring communication cycle. Within one communication cycle FlexRay offers the choice of two media access schemes. These are a static time division multiple access (TDMA) scheme, and a dynamic mini-slotting based scheme.

5.1.1 Communication cycle

The communication cycle is the fundamental element of the media access scheme within FlexRay. It is defined by means of a timing hierarchy.

The timing hierarchy consists of four timing hierarchy levels as depicted in Figure 5-1.

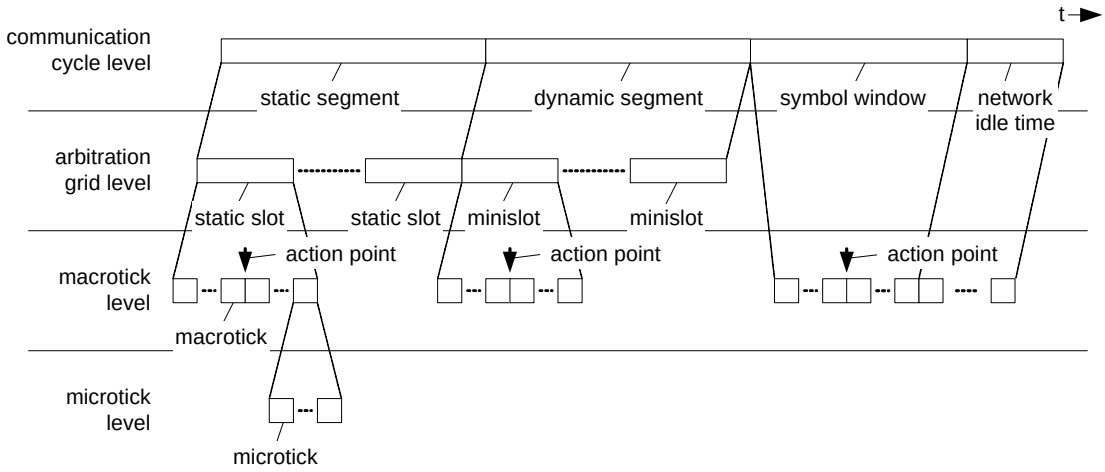

Figure 5-1: Timing hierarchy within the communication cycle.

The highest level, the communication cycle level, defines the communication cycle. It contains the static segment, the dynamic segment, the symbol window and the network idle time (NIT). Within the static segment a static time division multiple access scheme is used to arbitrate transmissions as specified in section 5.3.2. Within the dynamic segment a dynamic mini-slotting based scheme is used to arbitrate transmissions as specified in section 5.3.3. The symbol window is a communication period in which a symbol can be transmitted on the network as specified in section 5.3.4. The network idle time is a communication-free period that concludes each communication cycle as specified in section 5.3.5.

The next lower level, the arbitration grid level, contains the arbitration grid that forms the backbone of FlexRay media arbitration. In the static segment the arbitration grid consists of consecutive time intervals, called static slots, in the dynamic segment the arbitration grid consists of consecutive time intervals, called minislots.

The arbitration grid level builds on the macrotick level that is defined by the macrotick. The macrotick is specified in Chapter 8. Designated macrotick boundaries are called action points. These are specific instants at which transmissions shall start (in the static segment, dynamic segment and symbol window) and shall end (only in the dynamic segment).

The lowest level in the hierarchy is defined by the microtick, which is described in Chapter 8.

5.1.2 Communication cycle execution

Apart from during startup the communication cycle is executed periodically with a period that consists of a constant number of macroticks. The communication cycles are numbered from 0 to *gCycleCountMax*.

Arbitration within the static segment and the dynamic segment is based on the unique assignment of frame identifiers to the nodes in the cluster for each channel and a counting scheme that yields numbered transmission slots. The frame identifier determines the transmission slot and thus in which segment and when within the respective segment a frame shall be sent. The frame identifiers range from 1 to *cSlotIDMax*.

The communication cycle always contains a static segment. The static segment contains a configurable number *gNumberOfStaticSlots* of static slots. All static slots consist of an identical number of macroticks.

The communication cycle may contain a dynamic segment. The dynamic segment contains a configurable number *gNumberOfMinislots* of minislots. All minislots consist of an identical number of macroticks. If no dynamic segment is required it is possible to configure *gNumberOfMinislots* to zero minislots.

The communication cycle may contain a symbol window. The symbol window contains a configurable number *gdSymbolWindow* of macroticks. If no symbol window is required it is possible to configure *gdSymbolWindow* to zero macroticks.

The communication cycle always contains a network idle time. The network idle time contains the remaining number of macroticks within the communication cycle that are not allocated to the static segment, dynamic segment, or symbol window.

The constraints on the configuration of the communication cycle are defined in Appendix B.

Figure 5-2 illustrates the overall execution of the communication cycle.

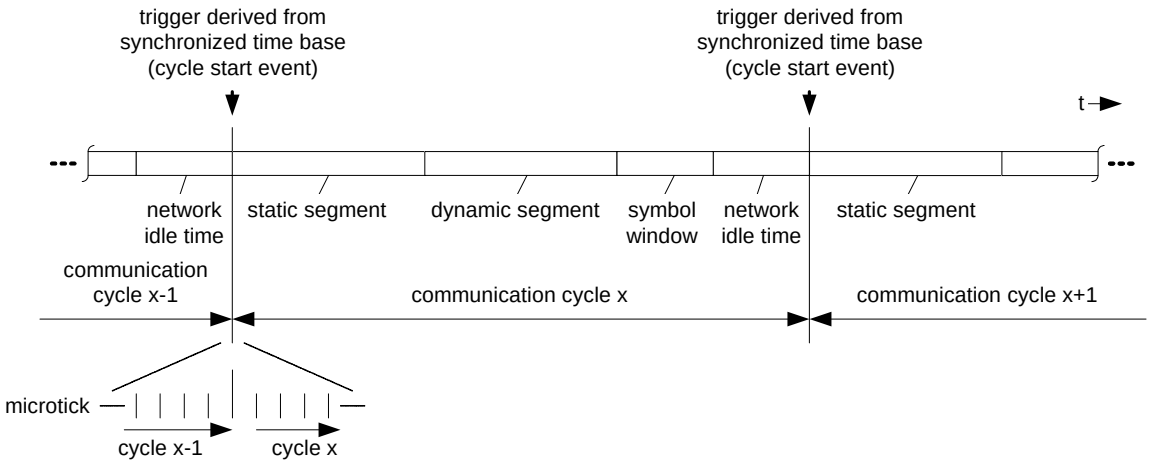

Figure 5-2: Time base triggered communication cycle.

The node shall maintain a cycle counter *vCycleCounter* that contains the number of the current communication cycle. Initialization and maintenance of the cycle counter are specified in Chapter 8.

The media access procedure is specified by means of the Media Access Control process for channel A. The node shall contain an equivalent Media Access Control process for channel B.

5.1.3 Static segment

Within the static segment a static time division multiple access scheme is applied to coordinate transmissions.

5.1.3.1 Structure of the static segment

In the static segment all communication slots are of identical, statically configured duration and all frames are of identical, statically configured length. For communication within the static segment the following constraints apply:

1. If a node has a key slot or key slots, that node shall transmit a frame in the key slot(s) on all connected channels in all communication cycles.
2. In slots other than the key slot(s), frames may be transmitted on either channel, or on both.
3. In a given communication cycle, no more than one node shall transmit a frame with a given frame ID on a given channel. It is allowed, however, for different nodes to transmit frames with the same frame ID on the same channel in different communication cycles.
4. If a non-sync node is configured to enter key slot mode after startup (i.e., *pKeySlotOnlyEnabled* is true) the node shall designate one frame as the key slot frame via the parameter *pKeySlotID*.

5.1.3.2 Execution and timing of the static segment

In order to schedule transmissions each node maintains a slot counter state variable *vSlotCounter* for channel A and a slot counter state variable *vSlotCounter* for channel B. Both slot counters are initialized with 1 at the start of each communication cycle and incremented at the end boundary of each slot.

Figure 5-3 illustrates all transmission patterns that are possible for a single node within the static segment. In slot 1 the node transmits a frame on channel A and a frame on channel B. In slot 2 the node transmits a frame only on channel A[70]. In slot 3 no frame is transmitted on either channel.

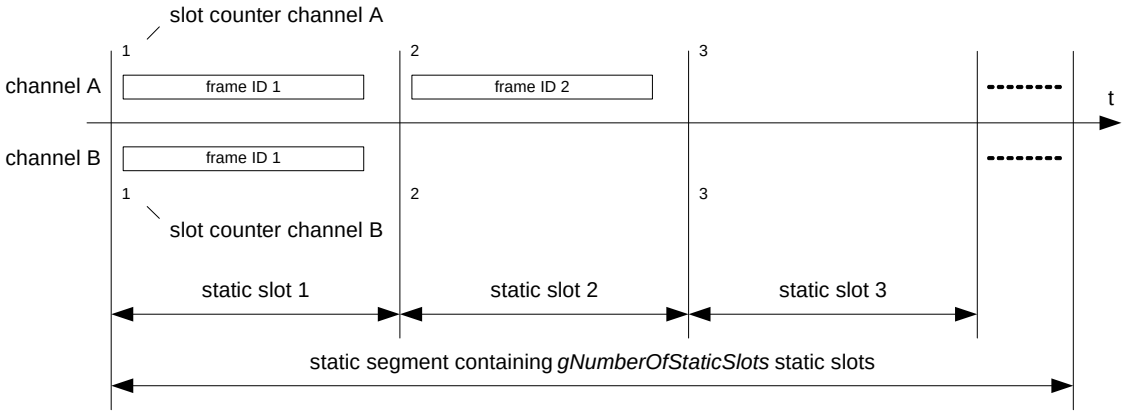

Figure 5-3: Structure of the static segment.

The number of static slots *gNumberOfStaticSlots* is a global constant for a given cluster.

All static slots consist of an identical number of *gdStaticSlot* macroticks. The number of macroticks per static slot *gdStaticSlot* is a global constant for a given cluster.

[70] Analogously, transmitting only on channel B is also allowed.

For any given node one or two[71] static slots (as defined in *pKeySlotID* and, in some cases, *pSecondKeySlotID*) may be assigned to contain sync frames (as identified by *pKeySlotUsedForSync*), a special type of frame required for synchronization within the cluster. Specific sync frames may be assigned to be startup frames (as identified by *pKeySlotUsedForStartup*).

Figure 5-4 depicts the detailed timing of the static slot.

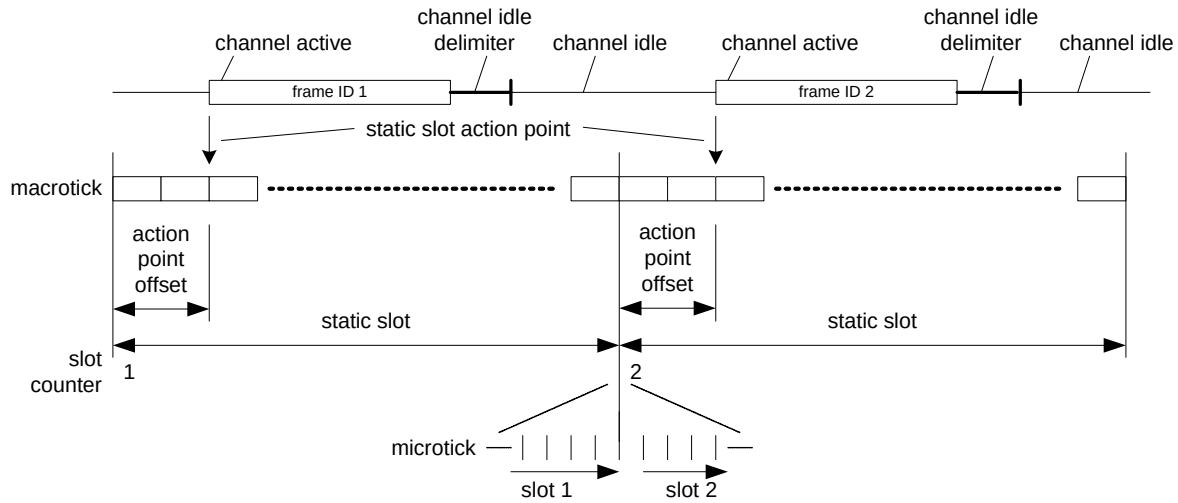

Figure 5-4: Timing within a static slot.

Each static slot contains an action point that is offset from the start of the slot by *gdActionPointOffset* macroticks. In the static segment frame transmissions start at the action point of the static slot. The number of macroticks contained in the action point offset *gdActionPointOffset* is a global constant for a given cluster.

5.1.4 Dynamic segment

Within the dynamic segment a dynamic mini-slotting based scheme is used to arbitrate transmissions.

5.1.4.1 Structure of the dynamic segment

In the dynamic segment the duration of communication slots may vary in order to accommodate frames of varying length. Frame lengths can be different for different slots in the same communication cycle, and can also be different for slots with the same identifier in different communication cycles.

For communication within the dynamic segment the following constraints apply:

1. Sync frames, startup frames, and null frames are not allowed.
2. In a given communication cycle, transmission of a given frame may be attempted on either channel, on both channels, or on neither channel[72]. Due to the independent nature of the channel-dependent media access processes it is possible that frame transmission that is attempted on both channels may be successful on one channel but unsuccessful on the other. It is also possible that a frame successfully transmitted on both channels is transmitted at different points in time on the different channels.

[71] Clusters using the TT-D synchronization mode have only one such slot, specified by *pKeySlotID*. Coldstart nodes of clusters using the TT-E or TT-L synchronization modes have two such slots, identified by *pKeySlotID* and *pSecondKeySlotID*.

[72] In the dynamic segment a node can choose not to transmit in a particular slot even if it is the assigned owner of the slot. This may be contrasted with the static segment, where a node must always send some type of frame (non-null or null) if it is the assigned owner of a slot.

3. In a given communication cycle, no more than one node shall transmit a frame with a given frame ID on a given channel. It is allowed, however, for different nodes to transmit frames with the same frame ID on the same channel in different communication cycles.
4. A node cannot transmit frames in the dynamic segment when operating in key slot only mode. As a consequence, a node's key slot frames (as determined by the parameter *pKeySlotID* or *pSecondKeySlotID*) cannot be sent in the dynamic segment.

5.1.4.2 Execution and timing of the dynamic segment

In order to schedule transmissions each node continues to maintain the two slot counters - one for each channel - throughout the dynamic segment. While the slot counters for channel A and for channel B are incremented simultaneously within the static segment, they may be incremented independently according to the dynamic arbitration scheme within the dynamic segment.

Figure 5-5 outlines the media access scheme within the dynamic segment. As illustrated in Figure 5-5, media access on the two communication channels may not necessarily occur simultaneously. Both communication channels do, however, use common arbitration grid timing that is based on minislots.

The number of minislots *gNumberOfMinislots* is a global constant for a given cluster.

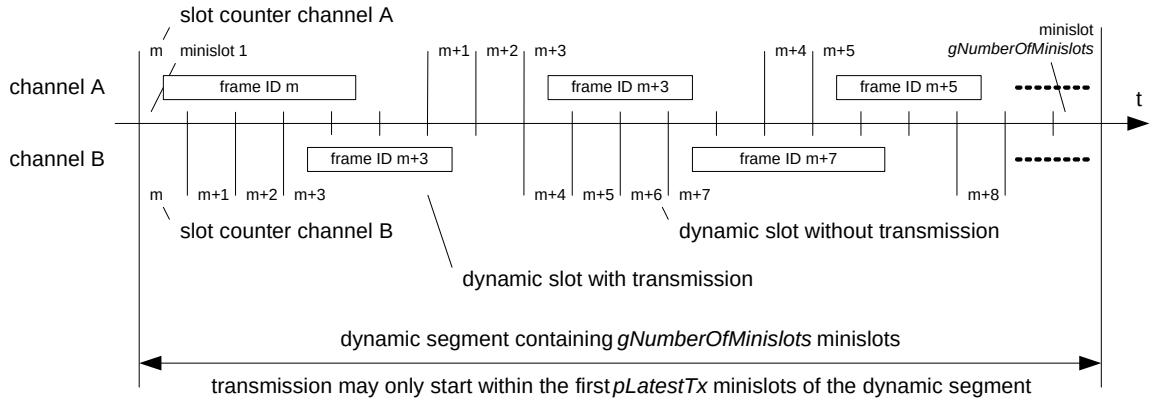

Figure 5-5: Structure of the dynamic segment.

Each minislot contains an identical number of *gdMinislot* macroticks. The number of macroticks per minislot *gdMinislot* is a global constant for a given cluster.

Within the dynamic segment a set of consecutive dynamic slots that contain one or multiple minislots are superimposed on the minislots. The duration of a dynamic slot depends on whether or not communication, i.e. frame transmission or reception, takes place. The duration of a dynamic slot is established on a per channel basis. Figure 5-5 illustrates how the duration of a dynamic slot adapts depending on whether or not communication takes place.

The node performs slot counting in the following way:

- The dynamic slot consists of one minislot if no activity takes place on the channel, i.e. the communication channel is in the channel idle state throughout the corresponding minislot.
- The dynamic slot consists of one or more minislots if activity takes place on the channel. If this activity is a legitimate frame transmission (as opposed to noise) the slot will consist of at least two minislots.

Each minislot contains an action point that is offset from the start of the minislot. With the possible exception of the first dynamic slot (explained below), this offset is *gdMinislotActionPointOffset* macroticks. The number

of macroticks within the minislot action point offset *gdMinislotActionPointOffset* is a global constant for a given cluster.

Figure 5-6 depicts the detailed timing of a minislot.

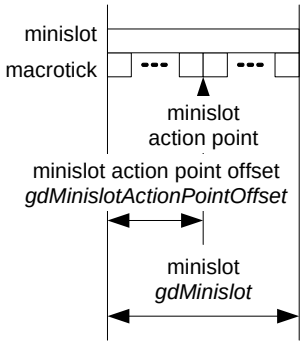

Figure 5-6: Timing within a minislot.

In the dynamic segment, frame transmissions start at the minislot action point of the first minislot of the corresponding dynamic slot. In the dynamic segment, frame transmissions also end at a minislot action point. This is achieved by means of the dynamic trailing sequence (DTS) as specified in Chapter 3.

In contrast to a static slot, the dynamic slot consists of two distinct phases - a mandatory dynamic slot transmission phase and an optional dynamic slot idle phase. The dynamic slot transmission phase extends from the start of the dynamic slot to the end of the last minislot in which the transmission terminates. The dynamic slot idle phase is an optional phase that extends from the end of the dynamic slot transmission phase to the end of the dynamic slot. Both the dynamic slot transmission phase and the dynamic slot idle phase (if it exists) consist of an integral number of minislots. The optional dynamic slot idle phase is defined as a communication-free phase that provides additional time to allow all nodes to complete idle detection within the dynamic slot. A dynamic slot idle phase is not always required - in some cases the time difference between the point at which transmission ends (which always occurs at a minislot action point) and the end of the minislot is sufficient to allow all nodes in the system to detect idle - such systems do not require a dynamic slot idle phase. In some cases, however, the time after the minislot action point would be insufficient to ensure that all nodes in the system can detect idle within the dynamic slot. In such systems, every dynamic slot actually used for transmission is extended by a dynamic slot idle phase, allowing all nodes to detect idle within the dynamic slot.

Figure 5-7 depicts the structure of a dynamic slot that makes use of a dynamic slot idle phase.

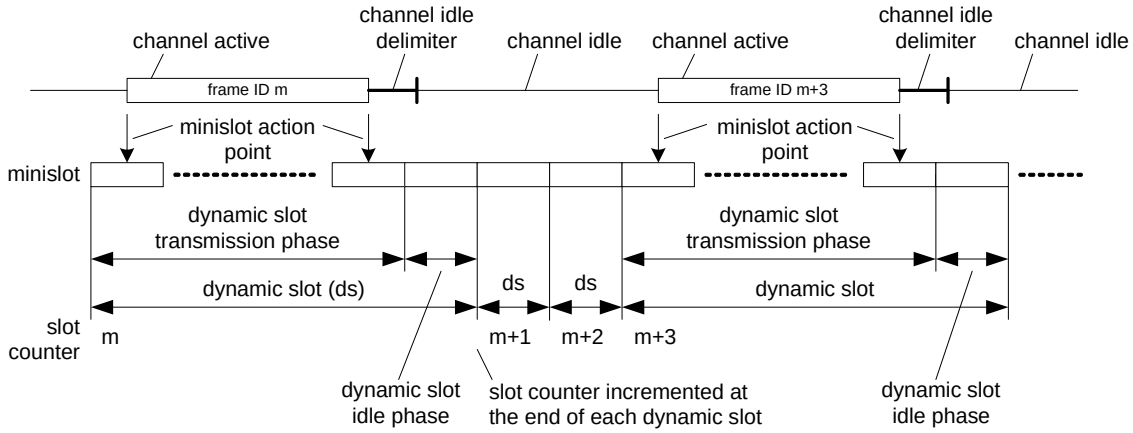

Figure 5-7: Structure of dynamic slots.

The start of the dynamic segment requires particular attention. The first action point in the dynamic segment occurs *gdActionPointOffset* macroticks after the end of the static segment if *gdActionPointOffset* is larger than *gdMinislotActionPointOffset* else it occurs *gdMinislotActionPointOffset* macroticks after the end of the static segment.[73]

The two cases are illustrated in Figure 5-8.

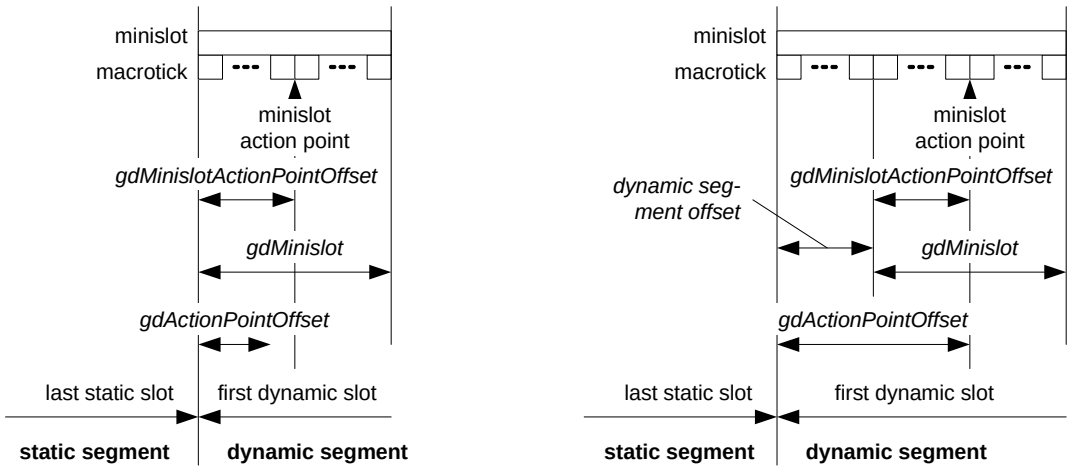

Figure 5-8: Timing at the boundary between the static and dynamic segments.

The node performs slot counter housekeeping on a per channel basis. At the end of every dynamic slot the node generally increments the slot counter *vSlotCounter* by one.[74] This is done until either

[73] This ensures that the duration of the gap following the last static frame transmission is at least as large as the gaps between successive frames within the static segment.

[74] Under special circumstances, the node may increase the slot counter by two to prevent a desynchronization of slot counters due to disturbances on the physical link.

1. the channel's slot counter has reached *cSlotIDMax*, or
2. the dynamic segment has reached the minislot *gNumberOfMinislots,* i.e. the end of the dynamic segment.

Once one of these conditions is met the node sets the corresponding slot counter to zero for the remainder of the communication cycle.

The arbitration procedure ensures that all fault-free receiving nodes implicitly know the dynamic slot in which the transmission starts. Further, all fault-free receiving nodes also agree implicitly on the minislot in which slot counting is resumed. As a result, the slot counters of all fault-free receiving nodes match the slot counter of the fault-free transmitting node and the frame identifier contained in the frame.

The arbitration of the dynamic segment relies heavily on the assumption that all *CE start*, *potential idle start* and *CHIRP* signals are generated in response to legitimate frame transmissions by other nodes. Should disturbances on the physical layer make it through the majority voting, this may be violated. As a consequence, it may occur that different nodes in the cluster have different notions of the slot counter value for a given minislot. Such a situation can also arise, for example, if the BD of a node is temporarily prevented from reception during a portion of the dynamic segment (by overtemperature or undervoltage conditions, for example). Dynamic segment desynchronization can also occur in certain situations where noise asymmetrically affects reception in the dynamic segment.[75] When slot counter desynchronization does occur in the dynamic segment its effect is limited to the specific instance of the dynamic segment. In other words, such desynchronization will not affect the operation of the static segment or the overall clock synchronization of the network, and will automatically be corrected at the end of the dynamic segment in which it occurred. The system designer should be aware of the possibility that slot counter desynchronization could occur in the dynamic segment.

The MAC has the ability to correct certain fault scenarios where activity caused by noise on the physical layer is shorter than *cFrameThreshold* bits and also short enough that both the start of the activity and the end of the idle detection following the activity both occur in a single minislot or within two adjacent minislots. If the MAC process detects such a noise event, it tries to revert to a state where it would have been if the noise event did not occur. This may cause an increment of the slot counter by two, or in some cases might even cause a slot counter increment during a subsequent reception (see Figure 5-21). To increase the robustness of the cluster, a node is prohibited from transmitting in the dynamic slot following a slot in which a noise event is detected.

5.1.5 Symbol window

Within the symbol window a single symbol, either an MTS or a WUDOP, may be sent. Arbitration among different senders is not provided by the protocol for the symbol window. If arbitration among multiple senders is required for the symbol window it has to be performed by means of a higher-level protocol.

Figure 5-9 outlines the media access scheme within the symbol window.

The number of macroticks per symbol window *gdSymbolWindow* is a global constant for a given cluster. The symbol window contains an action point that is offset from the start of the symbol window by *gdSymbolWindowActionPointOffset* macroticks. A symbol transmission starts at the action point within the symbol window.

[75] This can occur in some situations even though the MAC process in the dynamic segment has several mechanisms to prevent desynchronization when such noise exists.

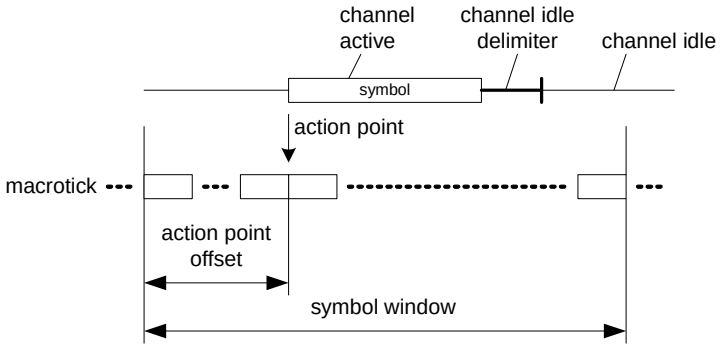

Figure 5-9: Timing within the symbol window.

5.1.6 Network idle time

The network idle time serves as a phase during which the node calculates and applies clock correction terms. Clock synchronization is specified in Chapter 8.

The network idle time also serves as a phase during which an implementation may perform various communication cycle related tasks.

The network idle time contains the remaining number of macroticks within the communication cycle not allocated to the static segment, dynamic segment, or symbol window.

5.2 Description

The relationship between the Media Access Control processes and the other protocol processes is depicted in Figure 5-10[76].

[76] The dark lines represent data flows between mechanisms that are relevant to this chapter. The lighter gray lines are relevant to the protocol, but not to this chapter.

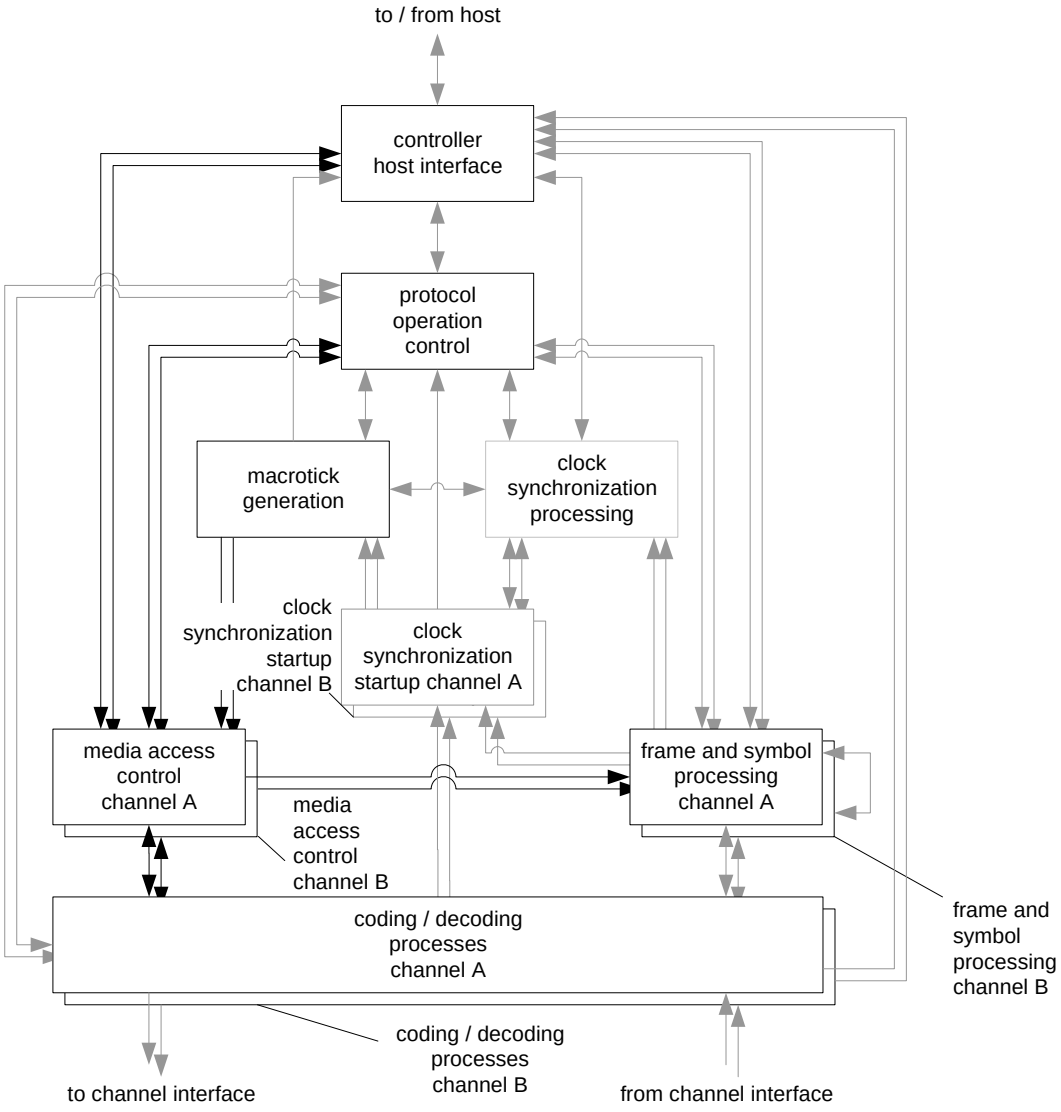

Figure 5-10: Media access control context.

In order to support two channels each node needs to contain a media access control process for channel A and a media access control process for channel B.

5.2.1 Operating modes

The protocol operation control process sets the operating mode of media access control for each communication channel:

1. In the STANDBY mode media access is effectively halted.
2. In the NOCE mode the media access process is executed, but no frames or symbols are sent on the channels.
3. In the STARTUPFRAMECAS mode transmissions are restricted to the transmission of one startup null frame per cycle on each configured channel in each key slot if the node is configured to send a startup frame. In addition the node sends an initial CAS symbol prior to the first communication cycle.

4. In the STARTUPFRAME mode transmissions are restricted to the transmission of one startup null frame per cycle on each configured channel in each key slot if the node is configured to send a startup frame.
5. In the KEYSLOTONLY mode the transmissions are restricted based on the synchronization type of the cluster and the role of the node. Sync nodes in a TT-D cluster have transmissions restricted to one sync frame per cycle on each confgured channel. Coldstart nodes in a TT-E or TT-L cluster have transmissons restricted to two startup frames per cycle on each configured channel. Non-sync nodes in any cluster type have transmissions restricted to a single specified key slot frame[77] per cycle on each configured channel.
6. In the ALL mode frames and symbols are sent in accordance with the node's transmission slot allocation.

Definition 5-1 gives the formal definition of the MAC operating modes.

```
newtype T_MacMode
      literals STANDBY, NOCE, STARTUPFRAMECAS, STARTUPFRAME, KEYSLOTONLY, ALL;
endnewtype;
```

Definition 5-1: Formal definition of T_MacMode.

5.2.2 Significant events

Within the context of media access control the node needs to react to a set of significant events. These are reception-related events, transmission-related events, and timing-related events.

5.2.2.1 Reception-related events

Figure 5-11 depicts the reception-related events that are significant for media access control.

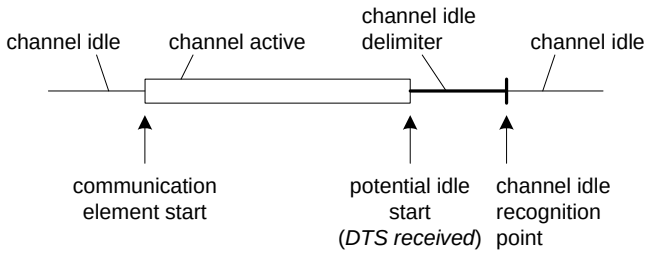

Figure 5-11: Reception-related events for MAC.

For communication channel A the reception-relevant events are:

1. communication element start on channel A (signal *CE start on A*, signal *idle end on A*),
2. potential idle start on channel A (signal *potential idle start on A*),
3. channel idle recognition point detected on channel A (signal *CHIRP on A*),
4. DTS high bit received on channel A (signal *DTS received on A*, only in the dynamic segment), and,
5. bit strobed on channel A (signal *bit strobed on A*, not shown in Figure 5-11).

For communication channel B the reception-relevant events are:

1. communication element start on channel B (signal *CE start on B*, signal *idle end on B*),
2. potential idle start on channel B (signal *potential idle start on B*),

[77] A node with *pKeySlotID* = 0 does not have a key slot, and thus will not transmit any frames when the MAC process is in the KEYSLOTONLY mode.

3. channel idle recognition point detected on channel B (signal *CHIRP on B*),
4. DTS high bit received on channel B (signal *DTS received on B*, only in the dynamic segment), and,
5. bit strobed on channel B (signal *bit strobed on B*, not shown in Figure 5-11).

Note that the channel-specific BITSTRB processes will output a *potential idle start* signal every time a bit strobed as high was preceded by a bit strobed as low. To keep the figure simple these signals are not shown in Figure 5-11 with the exception of the potential idle start at the end of channel activity.

5.2.2.2 Transmission-related events

Figure 5-12 depicts the transmission-related events that are significant for media access control.

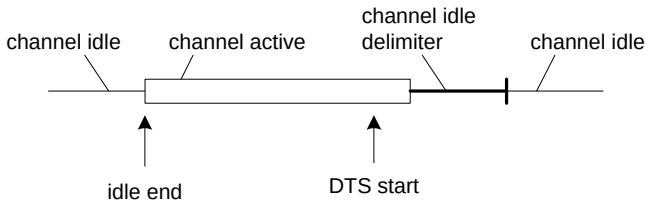

Figure 5-12: Transmission-related events for MAC.

For communication channel A the transmission-relevant events are:

1. the start of transmission ends the channel idle condition (signal *idle end on A*),
2. the start of the dynamic trailing sequence within the transmission pattern on channel A (signal *DTS start on A*).

For communication channel B the transmission-relevant events are:

1. the start of transmission ends the channel idle condition (signal *idle end on B*),
2. the start of the dynamic trailing sequence within the transmission pattern on channel B (signal *DTS start on B*).

5.2.2.3 Timing-related events

Both channels A and B are driven by the cycle start event that signals the start of each communication cycle (signal *cycle start (vCycleCounter)*; where *vCycleCounter* provides the number of the current communication cycle).

5.3 Media access control process

This section contains the formalized specification of the media access control process. The process is specified for channel A, the process for channel B is equivalent.

For each communication channel the MAC process contains the following states:

1. a *MAC:standby* state,
2. a *MAC:wait for CAS action point* state,
3. a *MAC:wait for the cycle start* state,
4. a *MAC:wait for the action point* state,
5. a *MAC:wait for the static slot boundary* state,
6. a *MAC:wait for the AP transmission start* state,
7. a *MAC:wait for the DTS start* state,

8. a *MAC:wait for the AP transmission end* state,
9. a *MAC:wait for the end of the dynamic segment* state,
10. a *MAC:wait for the end of the minislot* state,
11. a *MAC:wait for the end of activity* state,
12. a *MAC:wait for the end of the dynamic slot* state,
13. a *MAC:wait for the symbol window action point* state, and
14. a *MAC:wait for the end of the symbol window* state.

5.3.1 Initialization and *MAC:standby* state

Figure 5-13 depicts the specification of the media access process.

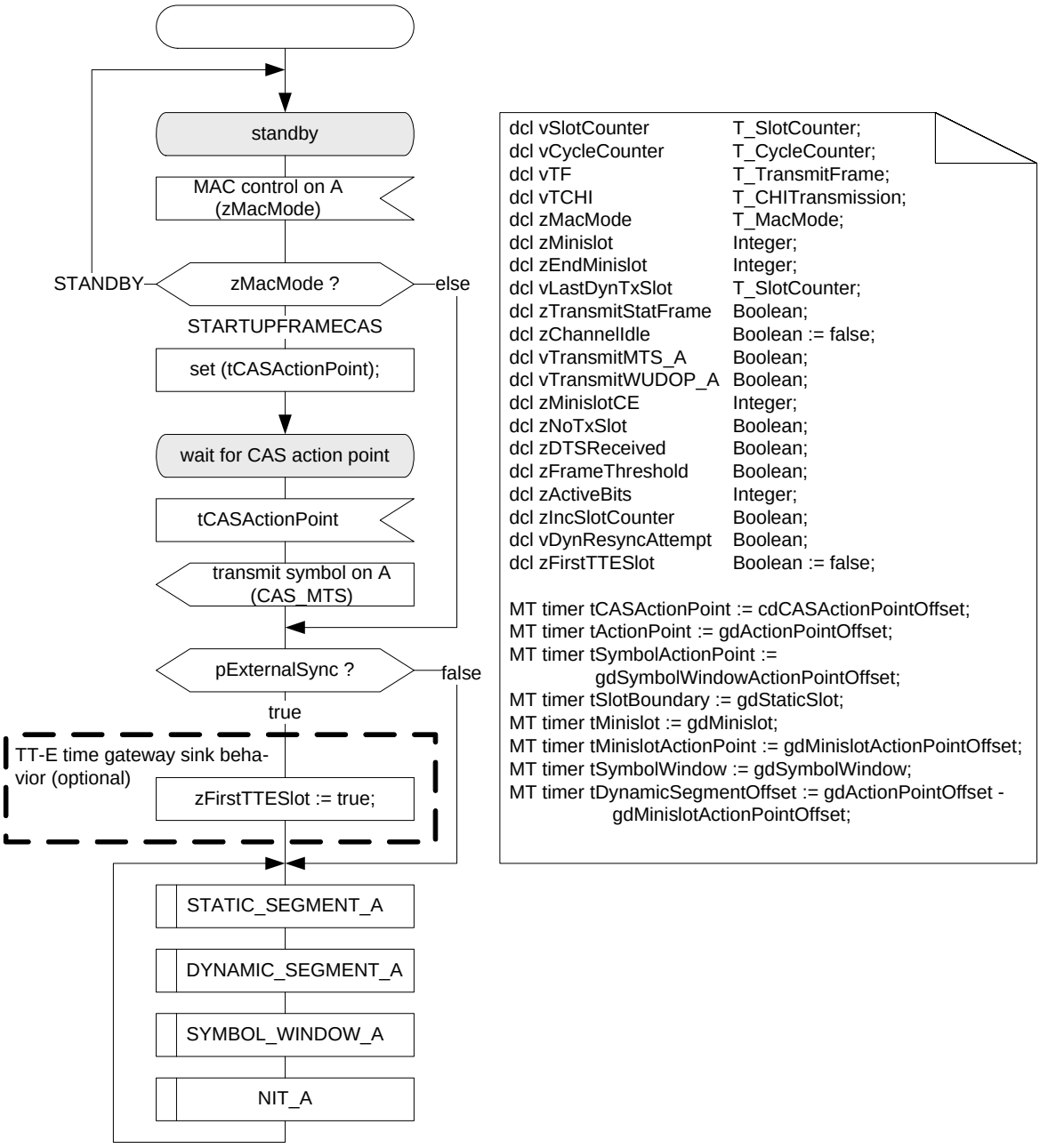

Figure 5-13: Media access process [MAC_A].

Figure 5-14 depicts how mode changes are processed by the media access process.

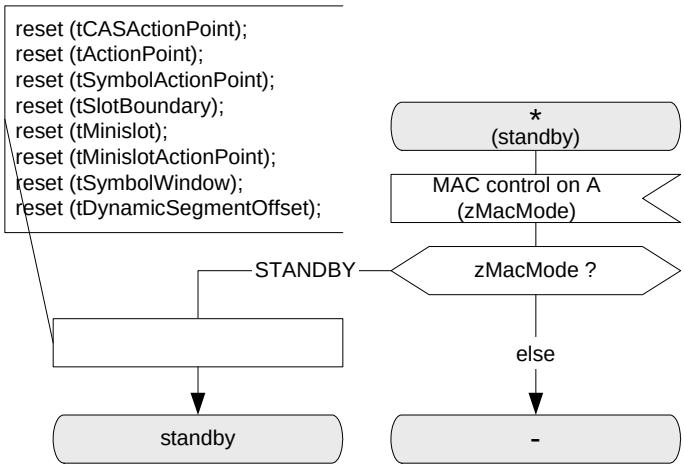

Figure 5-14: Media access control [MAC_A].

As depicted in Figure 5-15 a node shall terminate the MAC process upon occurrence of the terminate event issued by the protocol operation control.

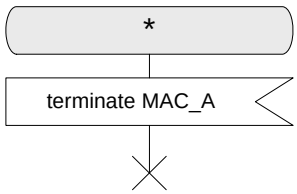

Figure 5-15: Termination of the MAC process [MAC_A].

5.3.2 Static segment related states

5.3.2.1 State machine for the static segment media access control

The node shall perform media access in the static segment as depicted in Figure 5-16.

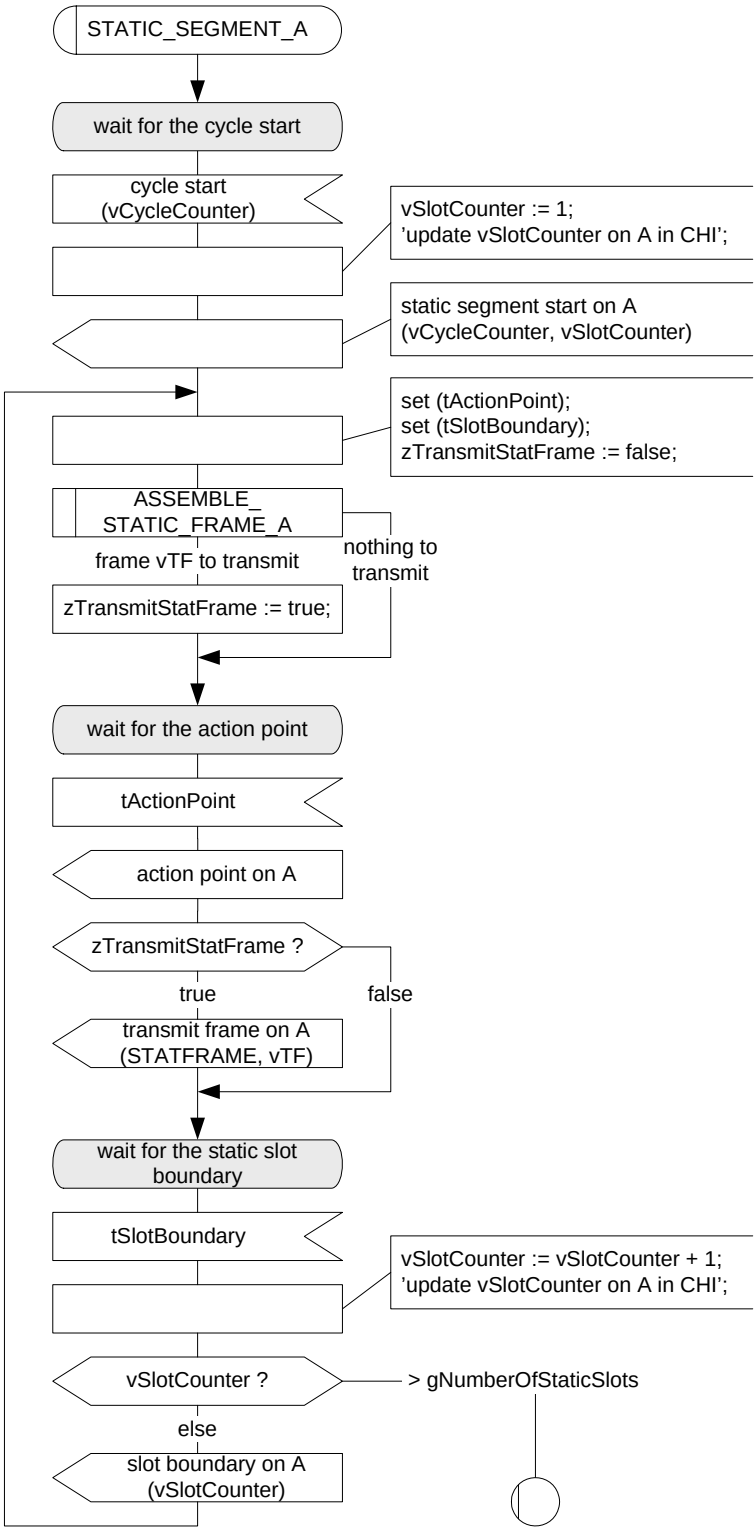

Figure 5-16: Media access in the static segment [MAC_A].

The node shall start frame transmission at the action point of an assigned static slot if appropriate transmission conditions are met (see section 5.3.2.2).

The transmission data that shall be sent is specified in the *T_TransmitFrame* data structure that is defined in Definition 3-2. Definition 5-2 defines *T_SlotCounter*.

```
syntype T_SlotCounter = Integer
      constants 0 : 2047
endsyntype;
```

Definition 5-2: Formal definition of T_SlotCounter.

At the end boundary of every static slot the node shall increment the slot counter *vSlotCounter* for channel A and the slot counter *vSlotCounter* for channel B by one.

5.3.2.2 Transmission conditions and frame assembly in the static segment

The node shall assemble a frame for transmission in the static segment according to the macro ASSEMBLE_STATIC_FRAME. The macro is depicted for channel A. Channel B is handled analogously.

In the static segment, whether or not a node shall transmit a frame depends on the current operating mode.

If media access is operating in the NOCE mode then the node shall transmit no frame.

If media access is operating in the STARTUPFRAMECAS mode or in the STARTUPFRAME mode then the node shall transmit a frame on each configured channel if the communication slot is a key slot.

If media access is operating in the ALL mode then the node shall transmit a frame on a channel if the slot is assigned to the node for the channel.

Data elements are imported from the CHI based on the channel, the current value of the slot counter, and the current value of the cycle counter. The CHI is assumed to return a data structure *T_CHITransmission* that is defined according to Definition 5-3.

```
newtype T_CHITransmission
struct
      Assignment              T_Assignment;
      TxMessageAvailable      Boolean;
      PPIndicator             T_PPIndicator;
      HeaderCRC               T_HeaderCRC;
      Length                  T_Length;
      Message                 T_Payload;
endnewtype;
```

Definition 5-3: Formal definition of T_CHITransmission.

```
newtype T_Assignment
      literals UNASSIGNED, ASSIGNED;
endnewtype;
```

Definition 5-4: Formal definition of T_Assignment.

Assuming a variable *vTCHI* of type *T_CHITransmission* imported from the CHI, the node shall assemble the frame in the following way if *vTCHI!Assignment* is set to ASSIGNED:

1. The reserved bit shall be set to zero.
2. If *vSlotCounter* equals *pKeySlotID*, or if *pTwoKeySlotMode* is true and *vSlotCounter* equals *pSecond-KeySlotID*, then
   a. the startup frame indicator shall be set in accordance with *pKeySlotUsedForStartup*, and
   b. the sync frame indicator shall be set in accordance with *pKeySlotUsedForSync*.

else

c. the startup frame indicator shall be set to zero, and

d. the sync frame indicator shall be set to zero.

3. The frame ID field shall be set to the current value of the slot counter *vSlotCounter*.
4. The length field shall be set to *gPayloadLengthStatic*.
5. The header CRC shall be set to the value *vTCHI!HeaderCRC* retrieved from the CHI.
6. The cycle count field shall be set to the current value of the cycle counter *vCycleCounter*.
7. If the host has data available (*vTCHI!TxMessageAvailable* set to true) then

   a. the null frame indicator shall be set to one, and

   b. the payload preamble indicator shall be set to the value *vTCHI!PPIndicator* imported from the CHI, and

   c. if *gPayloadLengthStatic* > *vTCHI!Length* then the *vTCHI!Length* number of two-byte payload words shall be copied from *vTCHI!Message* to *vTF!Payload*. The remaining (*gPayloadLengthStatic* - *vTCHI!Length*) two-byte payload words in *vTF!Payload* shall be set to the padding pattern 0x0000

      else if *gPayloadLengthStatic* = *vTCHI!Length* then *vTCHI!Length* number of two-byte payload words shall be copied from *vTCHI!Message* to *vTF!Payload*

      else the first *gPayloadLengthStatic* number of two-byte payload words shall be copied from *vTCHI!Message* to *vTF!Payload*.

else if the host has no data available (*vTCHI!TxMessageAvailable* set to false) then

   d. the null frame indicator shall be set to zero, and

   e. the payload preamble indicator shall be set to zero, and

   f. *gPayloadLengthStatic* number of two-byte payload words in *vTF!Payload* shall be set to the padding pattern 0x0000.

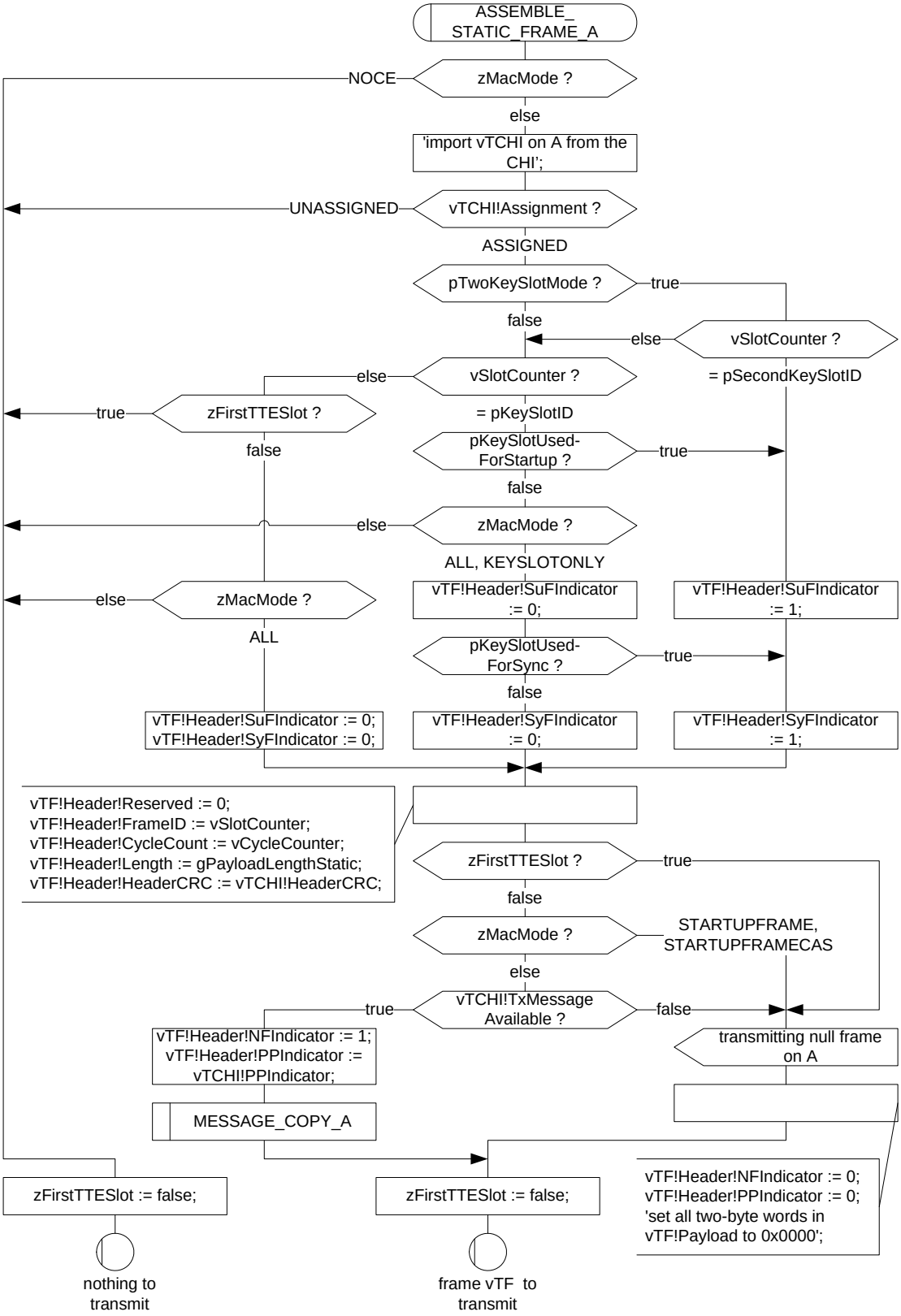

Figure 5-17: Frame assembly in the static segment [MAC_A].

The handling of a transmission in the first static slot in the first cycle following the startup of a TT-E coldstart node is different from the handling in subsequent slots and cycles.[78] In the first slot after a TT-E coldstart node's transition from the *POC:external startup* state to the *POC:normal active* state a TT-E coldstart node will only transmit a null frame, and only if this slot is a key slot, even if the media access is operating in the ALL mode. This is controlled by the variable *zFirstTTESlot*, which is set to true before the first cycle and set to false during the first slot. This variable enables the transmission of a null frame if the first slot is a key slot and disables transmission if the first slot is not a key slot.

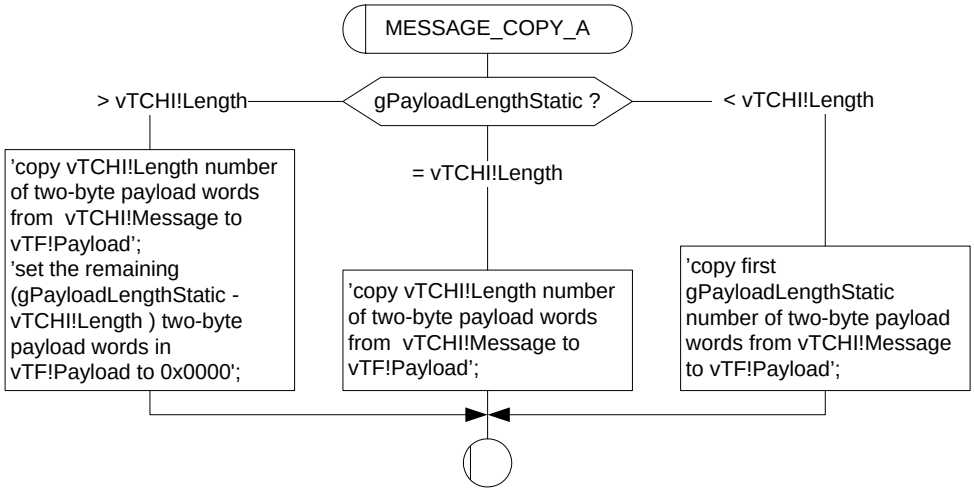

Figure 5-18: Message copying and padding in the static segment [MAC_A].

5.3.3 Dynamic segment related states

5.3.3.1 State machine for the dynamic segment media access control

The node shall perform media access in the dynamic segment as depicted in Figure 5-19 and subsequent Figures.

In the dynamic segment the node shall increment the slot counter *vSlotCounter* at the end of each dynamic slot. If the increment would cause *vSlotCounter* to exceed the maximum slot ID *cSlotIDMax* then *vSlotCounter* is instead set to zero and remains at this value until the end of the dynamic segment.

[78] The special handling in the first slot of the first cycle is necessary because of the short time between the cycle start signal from the time gateway source node and the start of the cycle in the time gateway sink node. This short time would make it difficult for a practical implementation to identify and select the buffer related to the first static slot. Limiting transmissions to null frames in the first slot, if this slot is a key slot, makes this task easier, and does not have a substantial impact on cluster startup.

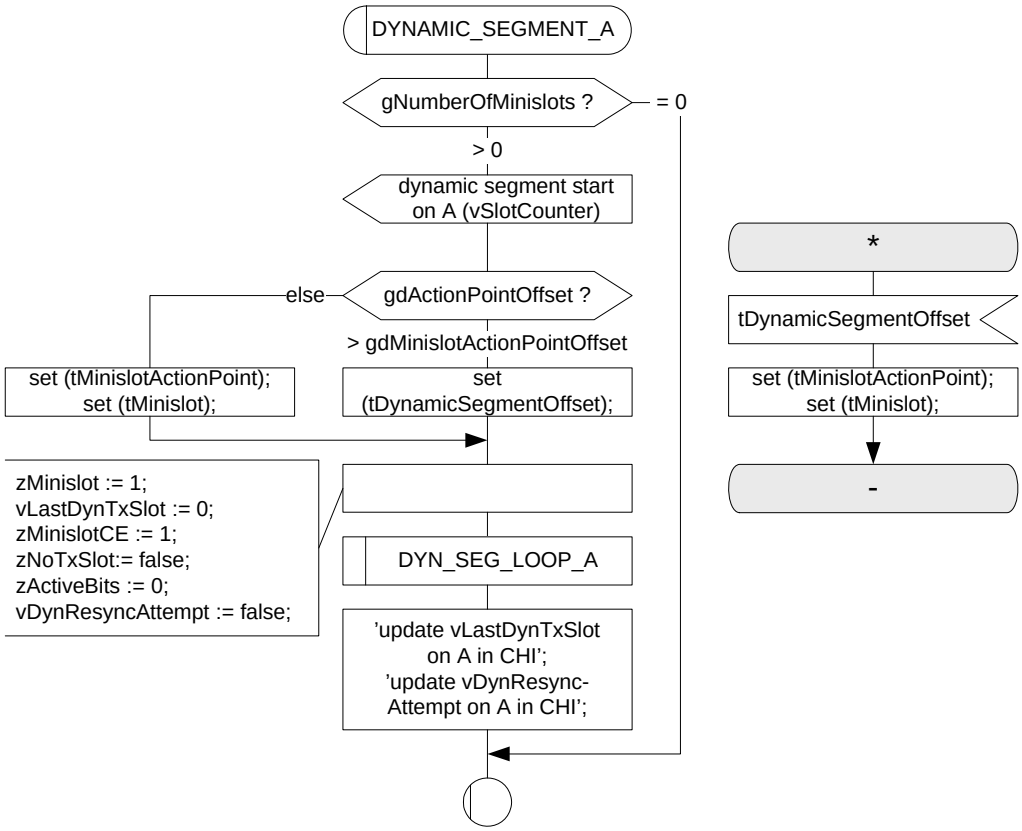

Figure 5-19: Media access in the dynamic segment [MAC_A].

The macro DYN_SEG_LOOP keeps track of the dynamic slot counter during the dynamic segment (see Figure 5-21).

At the start of each dynamic slot, the node checks whether there is still enough time left in the dynamic segment for a transmission or if no transmission is allowed in this dynamic slot due to the detection of a possible slot counter desynchronization (as indicated by the variable *zNoTxSlot*). If a transmission is allowed, the node determines whether it has the right and need to transmit in the current dynamic slot and, if yes, does so. The transmission is described by the TRANSMIT_DYNAMIC_FRAME macro (see Figure 5-22).

If the node does not transmit itself, it awaits transmissions of other nodes in the *MAC:wait for the end of the minislot* state. Should no *CE start* signal be detected before the end of the minislot, the node proceeds to the next dynamic slot. If a *CE start* signal is detected, the node notes the current minislot and starts to count the bits of the incoming communication element. The length of the incoming transmission is used as an indication of whether the incoming communication element is a dynamic frame or perhaps induced noise on the physical channel. Should the communication element end before the number of bits crosses the *cFrameThreshold*, the communication element is regarded as noise and the node tries to switch to a state where no noise was received. It does so by not applying the *gdDynamicSlotIdlePhase* lengthening of the dynamic slot on the one hand and by increasing the dynamic slot counter by two should a minislot boundary have occurred between the *CE start* signal and the *CHIRP* signal. The last *potential idle start* signal before the *CHIRP* signal marks the minislot in which the frame transmission ended, and is used to derive the last minislot of the dynamic slot. A fault-free frame reception will also enable the detection of the DTS, which is indicated by the CODEC process with the *DTS received* signal. As soon as the DTS was received, the node

locks down the end of the dynamic slot, with the intent that potential noise during the succeeding idle detection cannot affect the remaining dynamic slot length.

After the reception of the *CHIRP* signal, the node awaits the end of the dynamic slot. A *CE start* signal at this point in time is generally an indication of a fault on the bus; either the preceding or the current communication element was noise or a frame transmitted due to a fault condition. In case that the preceding element was already categorized as noise due to its short length, the node treats the new communication element as frame and potentially adjusts the dynamic slot counter. Under normal circumstances (i.e., in the noise-free case), no *CE start* signal will be received during the *MAC:wait for the end of the dynamic slot* state and the dynamic slot will end at the end of the minislot where *zMinislot* is equal to *zEndMinislot*. The end of the dynamic slot causes the dynamic slot counter to be incremented and then exported to the CHI. If the received communication element was shorter than the frame threshold *cFrameThreshold* and the dynamic slot was either one or two minislots long the node will abstain from transmitting in the following dynamic slot and a resynchronization attempt is noted for indication to the CHI at the end of the dynamic segment. If the received communication element was shorter than the frame threshold and the dynamic slot was two minislots long the dynamic slot counter is incremented twice instead of just once, as is normally the case.

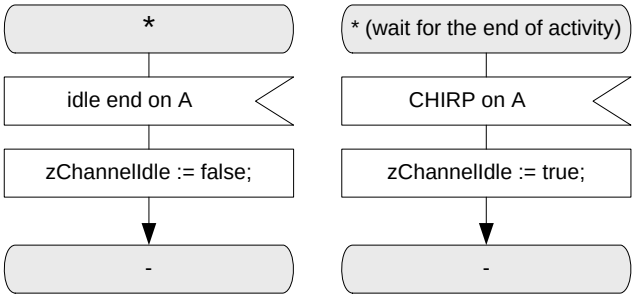

Figure 5-20: Channel idle tracking [MAC_A].

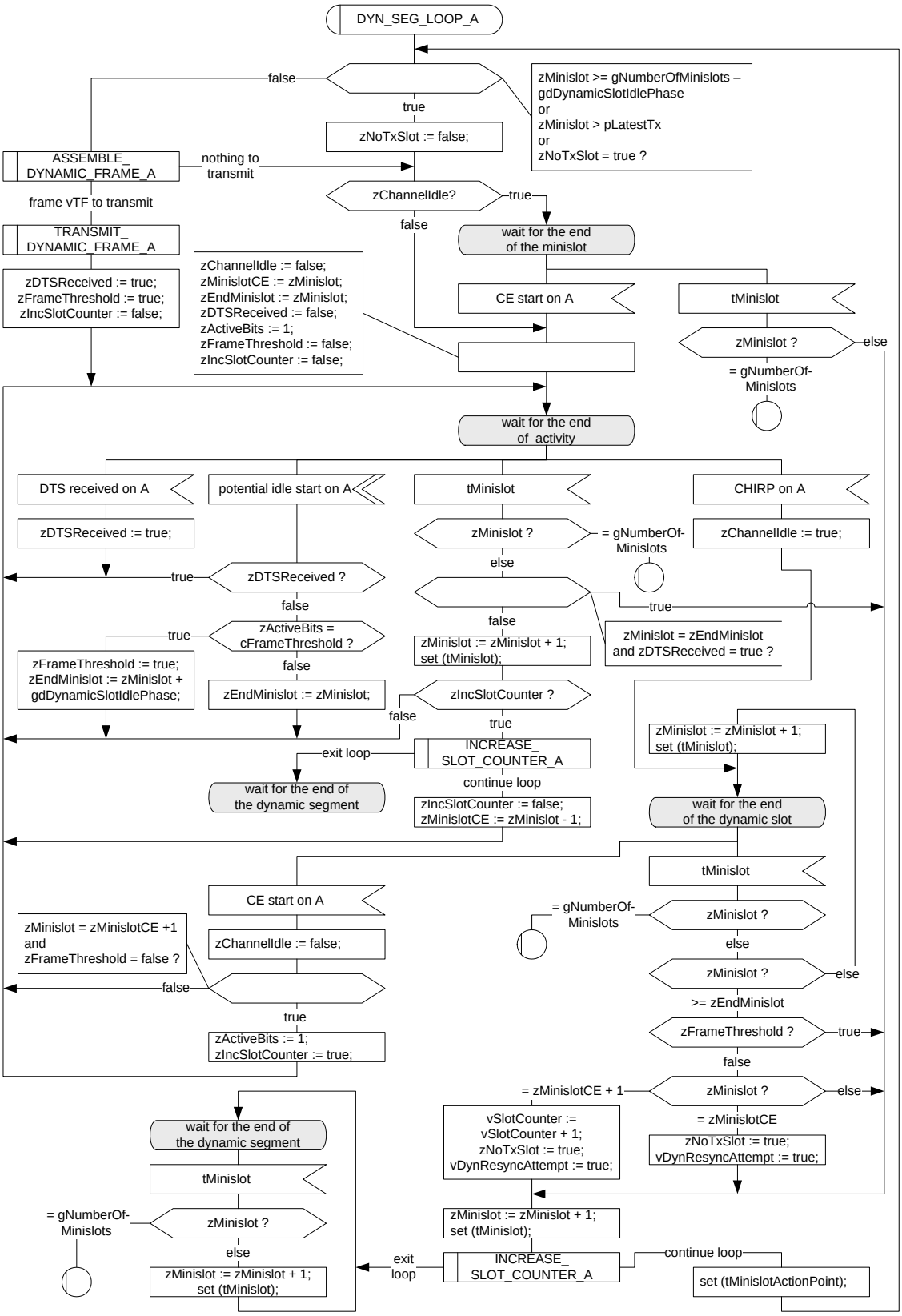

Figure 5-21: Media access in the dynamic segment arbitration [MAC_A].

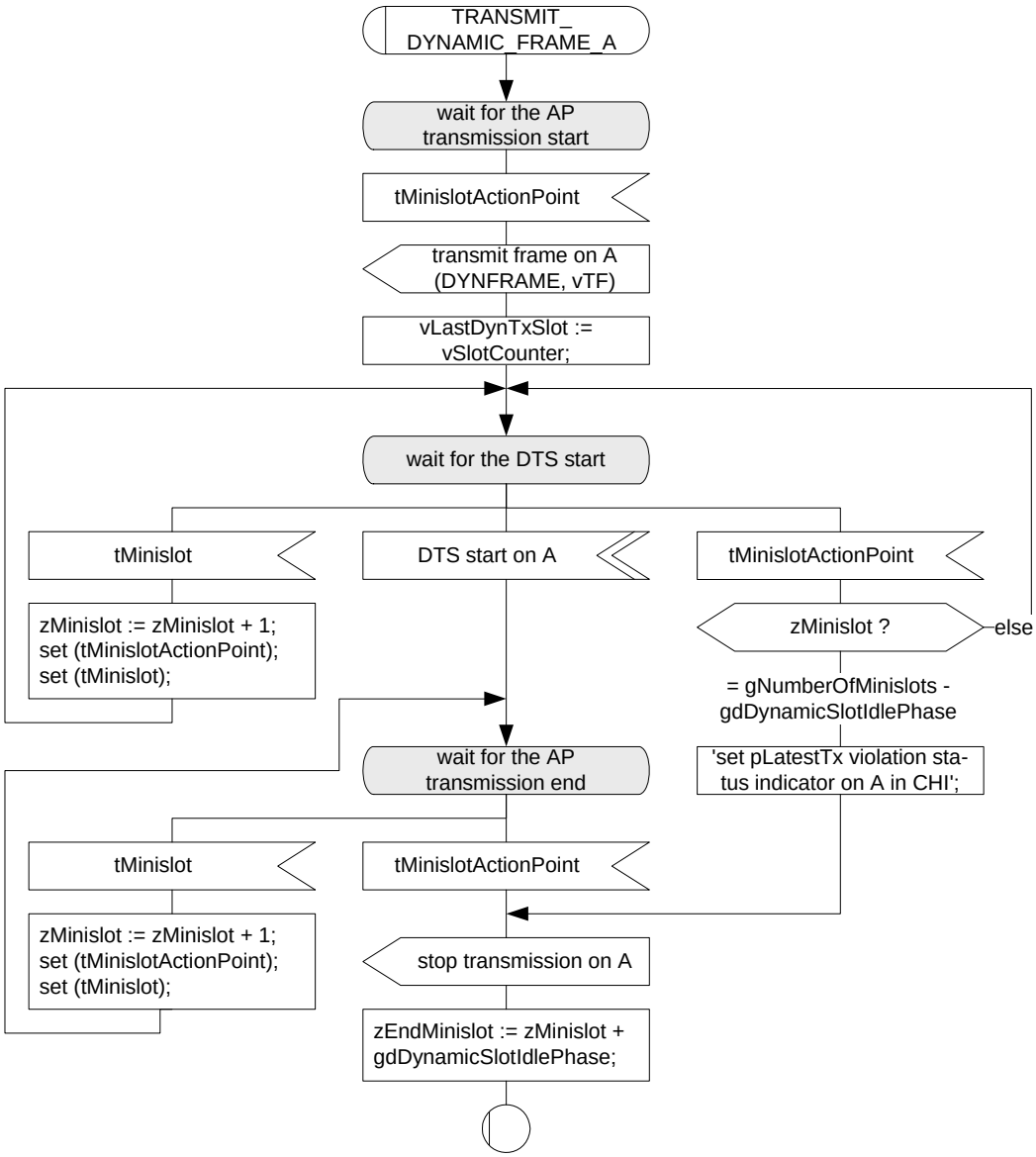

Figure 5-22: Transmission in the dynamic segment macro [MAC_A].

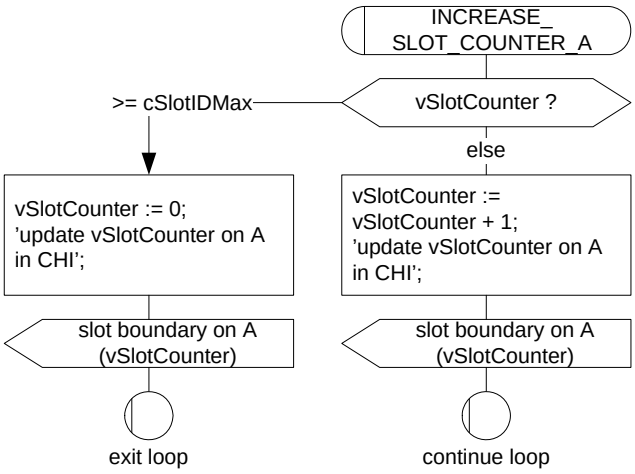

Figure 5-23: Slot counter increase macro [MAC_A].

The INCREASE_SLOT_COUNTER_A macro is called at the end of each dynamic slot to increase the dynamic slot counter and to communicate it to the CHI. If the dynamic slot counter were to surpass the maximum slot number *cSlotIDMax*, the slot counter shall not be incremented, but is instead reset to zero. The node shall then await the end of the dynamic segment.

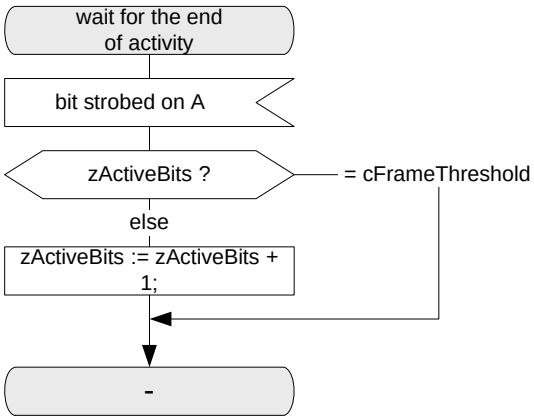

Figure 5-24: Counting of active bits [MAC_A].

To categorize an incoming communication element as noise, the node determines whether it is shorter than the frame threshold *cFrameThreshold*.

5.3.3.2 Transmission conditions and frame assembly in the dynamic segment

The node shall assemble a frame for transmission in the dynamic segment according to the macro ASSEMBLE_DYNAMIC_FRAME. The macro is depicted for channel A. Channel B is handled analogously.

In the dynamic segment, the node shall only transmit a frame on a channel if all following conditions are fulfilled:

- the media access is operating in the ALL mode,
- the current minislot number is not larger than *pLatestTx* minislot (a node-specific upper bound),

- the slot is assigned to the node,
- consistent payload data can be imported from the CHI,
- transmission is not prohibited in the slot by the *zNoTxSlot* variable (i.e., *zNoTxSlot* was not set to true in the previous slot), and
- there is at least a number of minislots equivalent to *gdDynamicSlotIdlePhase* before the end of the dynamic segment.

Assuming a variable *vTCHI* of type *T_CHITransmission* imported from the CHI the node shall assemble the frame in the following way if *vTCHI!Assignment* equals ASSIGNED and *vTCHI!TxMessageAvailable* equals true:

1. The reserved bit shall be set to zero.
2. The sync frame indicator shall be set to zero.
3. The startup frame indicator shall be set to zero.
4. The payload preamble indicator shall be set to the value *vTCHI!PPIndicator* retrieved from the CHI.
5. The frame ID field shall be set to the current value of the slot counter *vSlotCounter*.
6. The length field shall be set to *vTCHI!Length* retrieved from the CHI.
7. The header CRC shall be set to the value *vTCHI!HeaderCRC* retrieved from the CHI.
8. The cycle count field shall be set to the current value of the cycle counter *vCycleCounter*.
9. The null frame indicator shall be set to one.
10. *vTCHI!Length* number of two-byte payload words shall be copied from *vTCHI!Message* to *vTF!Payload*.

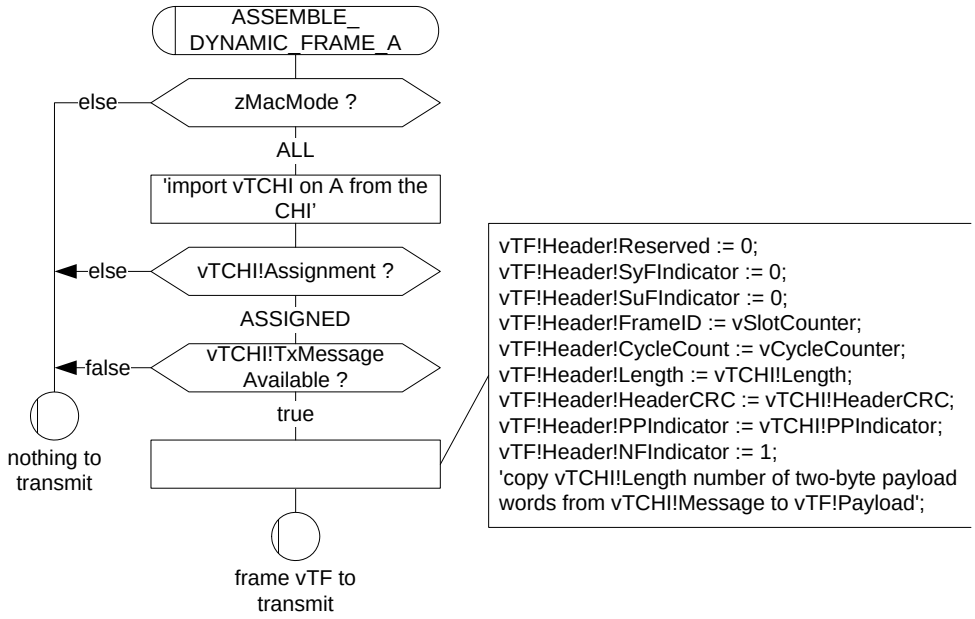

Figure 5-25: Frame assembly in the dynamic segment [MAC_A].

5.3.4 Symbol window related states

The node shall perform media access in the symbol window as depicted in Figure 5-26.
At the start of the symbol window the node shall set the slot counter *vSlotCounter* to zero. The node shall start symbol transmission at the action point of the symbol window if the media access is in the ALL mode and if a symbol is released for transmission.

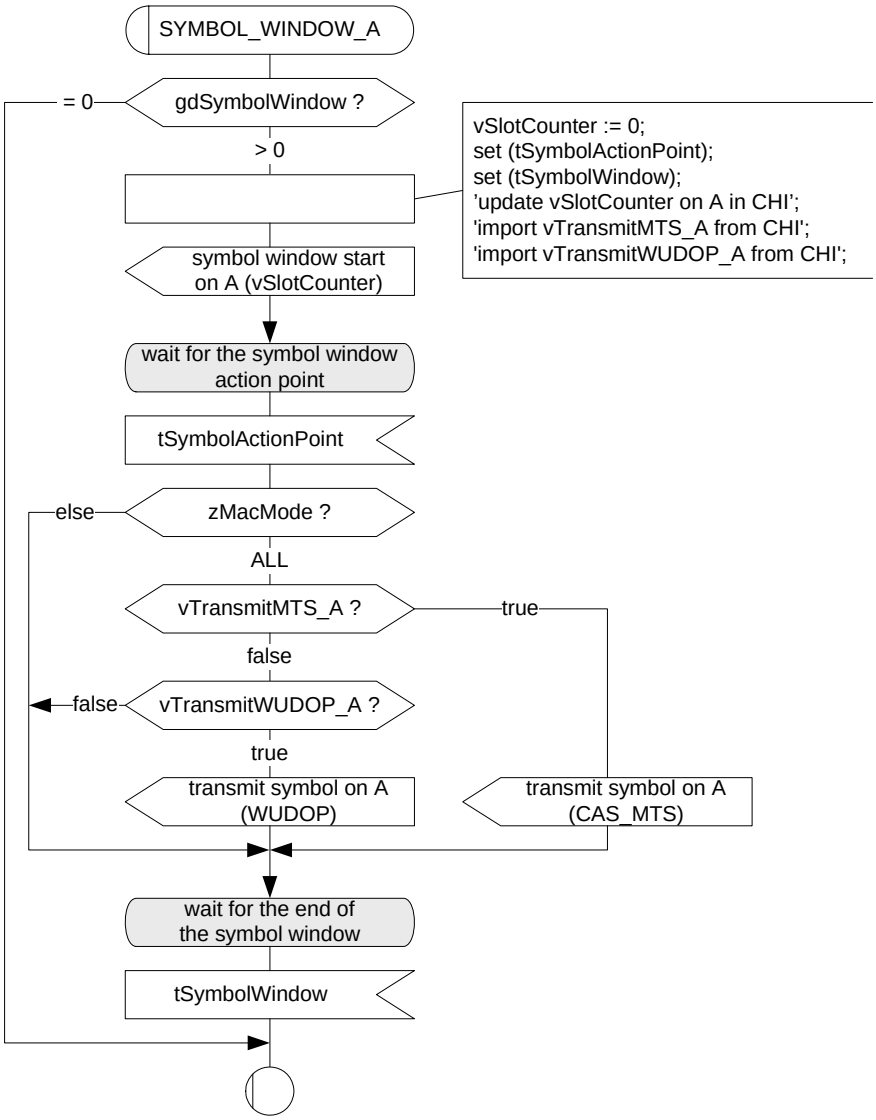

Figure 5-26: Media access in the symbol window [MAC_A].

5.3.5 Network idle time

Macro NIT in Figure 5-27 depicts the behavior at the start of the network idle time.

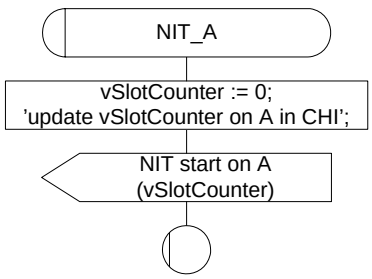

Figure 5-27: Network idle time [MAC_A].

At the start of the NIT the node shall set the slot counter *vSlotCounter* to zero.

Chapter 6
Frame and Symbol Processing

This chapter defines how the node shall perform frame and symbol processing.

6.1 Principles

Frame and symbol processing (FSP) is the main processing layer between frame and symbol decoding, which is specified in Chapter 3, and the controller host interface, which is specified in Chapter 9.

Frame and symbol processing checks the correct timing of frames and symbols with respect to the TDMA scheme, applies further syntactical tests to received frames, and checks the semantic correctness of received frames.

6.2 Description

The relationship between the Frame and Symbol Processing processes and the other protocol processes is depicted in Figure 6-1[79].

[79] The dark lines represent data flows between mechanisms that are relevant to this chapter. The lighter gray lines are relevant to the protocol, but not to this chapter.

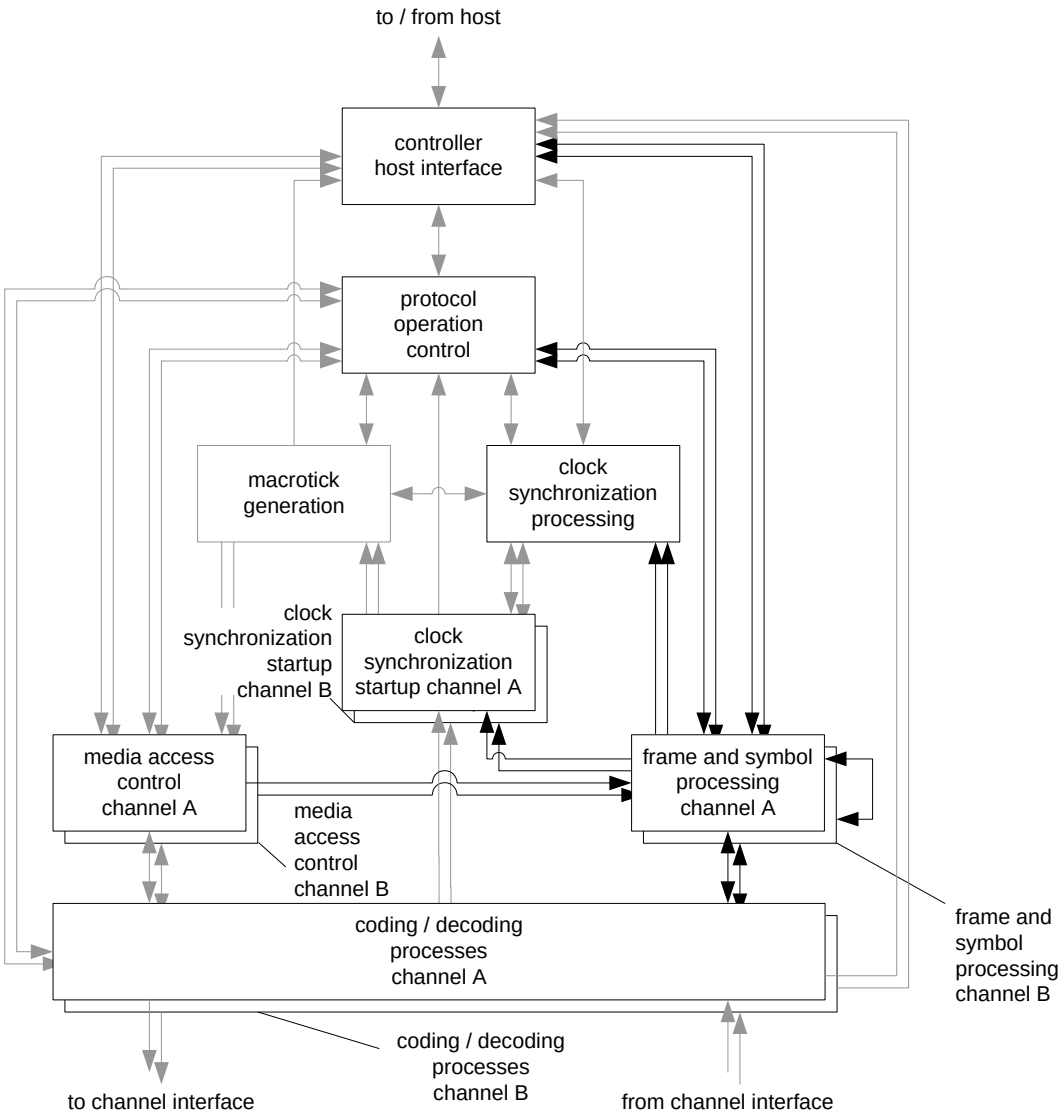

Figure 6-1: Frame and symbol processing context.

In order to support two channels each node needs to contain a frame and symbol processing process for channel A and a frame and symbol processing process for channel B.

6.2.1 Operating modes

The protocol operation control process sets the operating mode of frame and symbol processing for each communication channel:

1. In the STANDBY mode the execution of the frame and symbol processing process shall be halted.
2. In the STARTUP mode the frame and symbol processing process shall be executed but no update of the CHI takes place, except for decoded wakeup patterns.
3. In the GO mode the frame and symbol processing process shall be executed and the update of the CHI takes place.

Definition 6-1 gives the formal definition of the FSP operating modes.

```
newtype T_FspMode
```

```
      literals STANDBY, STARTUP, GO;
endnewtype;
```

Definition 6-1: Formal definition of T_FspMode.

6.2.2 Significant events

Within the context of frame and symbol processing the node needs to react to a set of significant events. These are reception-related events, decoding-related events, and timing-related events.

6.2.2.1 Reception-related events

Figure 6-2 depicts the reception-related events that are significant for frame and symbol processing.

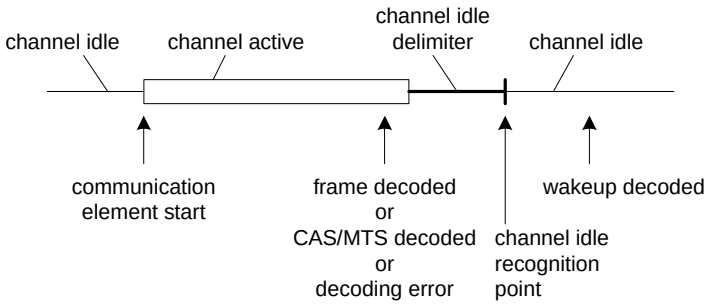

Figure 6-2: Reception-related events for FSP.

For communication channel A the reception-related events are

1. communication element start on channel A (signal *CE start on A*),
2. frame decoded on channel A (signal *frame decoded on A (vRF)*, where *vRF* provides the timestamp of the primary time reference point and the header as well as the payload of the received frame as defined in Definition 3-10),
3. CAS or MTS decoded on channel A (signal *CAS_MTS decoded on A*),
4. decoding error on channel on A (signal *decoding error on A (zDecodingError)*),
5. channel idle recognition point detected on channel A (signal *CHIRP on A*),
6. content error on channel B (signal *content error on B*)[80],
7. wakeup decoded on channel A (signal *wakeup decoded on A*).

For communication channel B the reception-related events are

1. communication element start on channel B (signal *CE start on B*),
2. frame decoded on channel B (signal *frame decoded on B (vRF)*, where *vRF* provides the timestamp of the primary time reference point and the header as well as the payload of the received frame as defined in Definition 3-10),
3. CAS or MTS decoded on channel B (signal *CAS_MTS decoded on B*),
4. decoding error on channel B (signal *decoding error on B (zDecodingError)*),
5. channel idle recognition point detected on channel B (signal *CHIRP on B*),
6. content error on channel A (signal *content error on A*),
7. wakeup decoded on channel B (signal *wakeup decoded on B*).

Definition 3-10 gives the formal definition of the *T_ReceiveFrame* data structure.

[80] In order to address channel consistency checks for sync frames.

6.2.2.2 Decoding-related events

For communication channel A the decoding-related events are

1. decoding halted on channel A (signal *decoding halted on A*),
2. decoding started on channel A (signal *decoding started on A*).

For communication channel B the decoding-related events are

1. decoding halted on channel B (signal *decoding halted on B*),
2. decoding started on channel B (signal *decoding started on B*).

6.2.2.3 Timing-related events

Figure 6-3 depicts the timing-related events that are significant for frame and symbol processing.

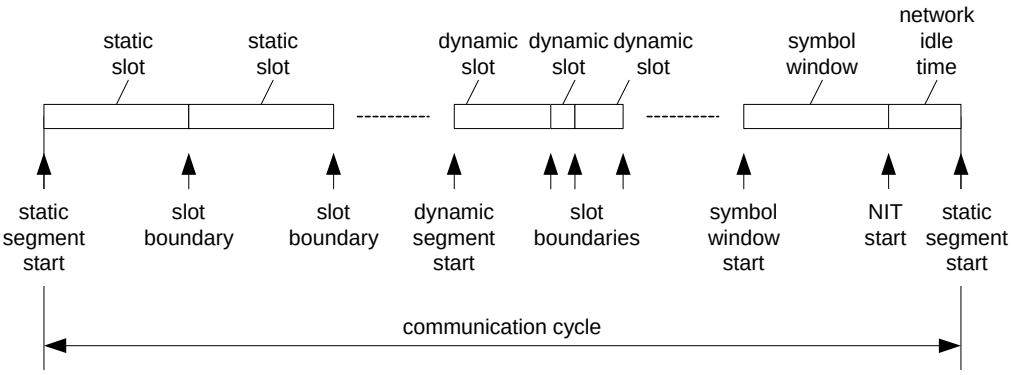

Figure 6-3: Timing-related events for FSP.

For communication channel A the relevant events are

1. static segment start on channel A (signal *static segment start on A (vCycleCounter, vSlotCounter)*; where *vCycleCounter* holds the number of the current communication cycle and *vSlotCounter* holds the number of the communication slot that is just beginning on channel A),
2. slot boundary on channel A (signal *slot boundary on A (vSlotCounter)*; where *vSlotCounter* holds the number of the communication slot that is just beginning on channel A),
3. dynamic segment start on channel A (signal *dynamic segment start on A (vSlotCounter)*; where *vSlotCounter* holds the number of the current communication slot on channel A),
4. symbol window start on channel A (signal *symbol window start on A (vSlotCounter)*; where *vSlotCounter* holds the value 0),
5. network idle time (NIT) start on channel A (signal *NIT start on A (vSlotCounter)*; where *vSlotCounter* holds the value 0).

For communication channel B the relevant events are

1. static segment start on channel B (signal *static segment start on B (vCycleCounter, vSlotCounter)*; where *vCycleCounter* holds the number of the current communication cycle and *vSlotCounter* holds the number of the communication slot that is just beginning on channel B),
2. slot boundary on channel B (signal *slot boundary on B (vSlotCounter)*; where *vSlotCounter* holds the number of the communication slot that is just beginning on channel B),
3. dynamic segment start on channel B (signal *dynamic segment start on B (vSlotCounter)*; where *vSlotCounter* holds the number of the current communication slot on channel B),

4. symbol window start on channel B (signal *symbol window start on B (vSlotCounter)*; where *vSlotCounter* holds the value 0),
5. network idle time (NIT) start on channel B (signal *NIT start on B (vSlotCounter)*; where *vSlotCounter* holds the value 0).

6.2.3 Status data

For each communication channel the node shall provide a slot status that is updated in the CHI as specified in Chapter 9.

Definition 6-2 gives the formal definition of the slot status.

```
newtype T_SlotStatus
struct
      Channel           T_Channel;
      SlotCount         T_SlotCounter;
      CycleCount        T_CycleCounter;
      ValidFrame        Boolean;
      ValidMTS          Boolean;
      SyntaxError       Boolean;
      ContentError      Boolean;
      BViolation        Boolean;
      FrameSent         Boolean;
      TxConflict        Boolean;
      NFIndicator       T_NFIndicator;
      Segment           T_Segment;
endnewtype;
```

Definition 6-2: Formal definition of T_SlotStatus.

The slot status consists of the following elements:

- *Channel* identifies the corresponding channel of the other slot status elements.
- *SlotCount* holds the value of the slot counter of the corresponding slot.
- *CycleCount* holds the value of the cycle counter of the corresponding cycle.
- *ValidFrame* denotes whether a valid frame was received in a slot of the static or dynamic segment. The element is set to false if no valid frame was received (or if a frame was transmitted in the slot), or to true if a valid frame was received.
- *ValidMTS* denotes whether a valid MTS was received in the symbol window. The element is set to false if no valid MTS was received, or to true if a valid MTS was received.
- *SyntaxError* denotes whether a syntax error has occurred. A syntax error occurs if any of the following criteria are met:

  1. the node starts transmitting while the channel is not in the idle state,
  2. a decoding error occurs,
  3. a frame is decoded in the symbol window or in the network idle time,
  4. a CAS/MTS is decoded in the static segment, in the dynamic segment, or in the network idle time,
  5. a frame is received within the slot after the reception of a semantically correct frame,
  6. an otherwise valid frame is received before transmission in a slot that is used for transmission,
  7. a syntactically valid frame is detected after transmission in a slot that is used for transmission.

This element is set to false if no syntax error occurred, or to true if a syntax error did occur. Note, it is possible to have *SyntaxError* = true and *ValidFrame* = true. This could occur, for example, if a syntactically incorrect frame is received first, followed by a semantically correct and syntactically correct frame in the same slot. This would result in *ValidFrame* = true, *SyntaxError* = true, and *ContentError* = false.

- *ContentError* denotes whether a content error has occurred. A content error occurs if any of the following criteria are met:

  1. in the static segment the header length contained in the header of the received frame does not match the stored header length in *gPayloadLengthStatic*,
  2. in the static segment the startup frame indicator contained in the header of the received frame is set to one while the sync frame indicator is set to zero,
  3. in the static segment the null frame indicator contained in the header of the received frame is set to zero and the payload preamble indicator is set to one,
  4. in the static or in the dynamic segment the frame ID contained in the header of the received frame does not match the current value of the slot counter or the frame ID equals zero in the dynamic segment,
  5. in the static or in the dynamic segment the cycle count contained in the header of the received frame does not match the current value of the cycle counter,
  6. in the dynamic segment the sync frame indicator contained in the header of the received frame is set to one,
  7. in the dynamic segment the startup frame indicator contained in the header of the received frame is set to one,
  8. in the dynamic segment the null frame indicator contained in the header of the received frame is set to zero.

  This element is set to false if no content error occurred, or to true if a content error did occur. Note - it is possible to have *ContentError* = true and *ValidFrame* = true. This could occur, for example, if a syntactically correct but semantically incorrect frame is received first, followed by a semantically correct and syntactically correct frame in the same slot. This would result in *ValidFrame* = true, *SyntaxError* = false, and *ContentError* = true.

- *BViolation* denotes whether a boundary violation occurred at either beginning or end of the corresponding slot. A boundary violation occurs if the node does not consider the channel to be idle at the boundary of a slot. The element is set to false if no boundary violation occurred.
  See Table 9-2 for a description of the meaning of various combinations of *ValidFrame*, *SyntaxError*, *ContentError*, and *BViolation*.
- *FrameSent* denotes whether a non-null frame was completely transmitted in the corresponding slot. The element is set to true if there was a complete non-null frame transmission in the slot, and otherwise it is set to false.
- *TxConflict* denotes whether reception was ongoing at the time the node started a transmission. The element is set to false if reception was not ongoing, or to true if reception was ongoing.
- *NFIndicator* denotes whether a null frame was received. The element is set to 0 if either a null frame, an invalid frame or nothing was received, or to 1 if a valid non-null frame was received.[81]
- *Segment* denotes the segment in which the slot status was recorded. Its type is defined in Definition 6-3.

```
newtype T_Segment
      literals STUP, STATIC, DYNAMIC, SW, NIT;
```

[81] The parameter *NFIndicator* is always set to 1 for valid frames in the dynamic segment.

```
endnewtype;
```

Definition 6-3: Formal definition of T_Segment.

The element *Segment* is set to STUP during the non-synchronized startup phase. The values STATIC, DYNAMIC, SW, and NIT denote the static segment, the dynamic segment, the symbol window, and the network idle time, respectively.

6.3 Frame and symbol processing process

This section contains the formalized specification of the frame and symbol processing process. The process is specified for channel A, the process for channel B is equivalent.

Figure 6-4 gives an overview of the frame and symbol processing-related state diagram.

For each communication channel the FSP process contains five states:

1. an *FSP:standby* state,
2. an *FSP:wait for CE start* state,
3. an *FSP:decoding in progress* state,
4. an *FSP:wait for CHIRP* state, and
5. an *FSP:wait for transmission end* state.

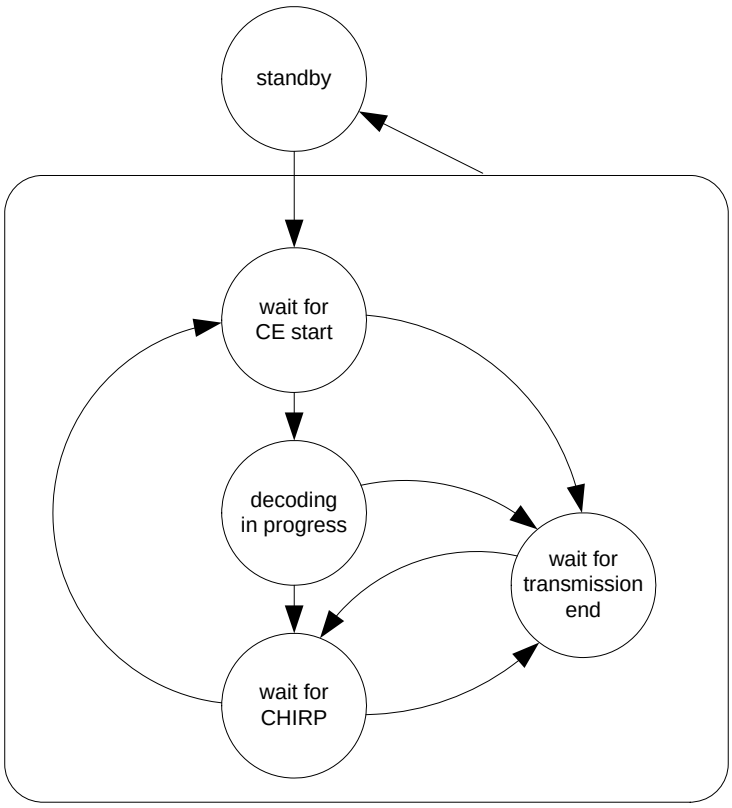

Figure 6-4: State overview of the FSP state machine (shown for one channel).

6.3.1 Initialization and *FSP:standby* state

As depicted in Figure 6-5, the node shall initially enter the *FSP:standby* state of the FSP process and wait for an FSP mode change initiated by the protocol operation control process.

A node shall leave the *FSP:standby* state if the protocol operation control process sets the FSP mode to STARTUP or to GO.

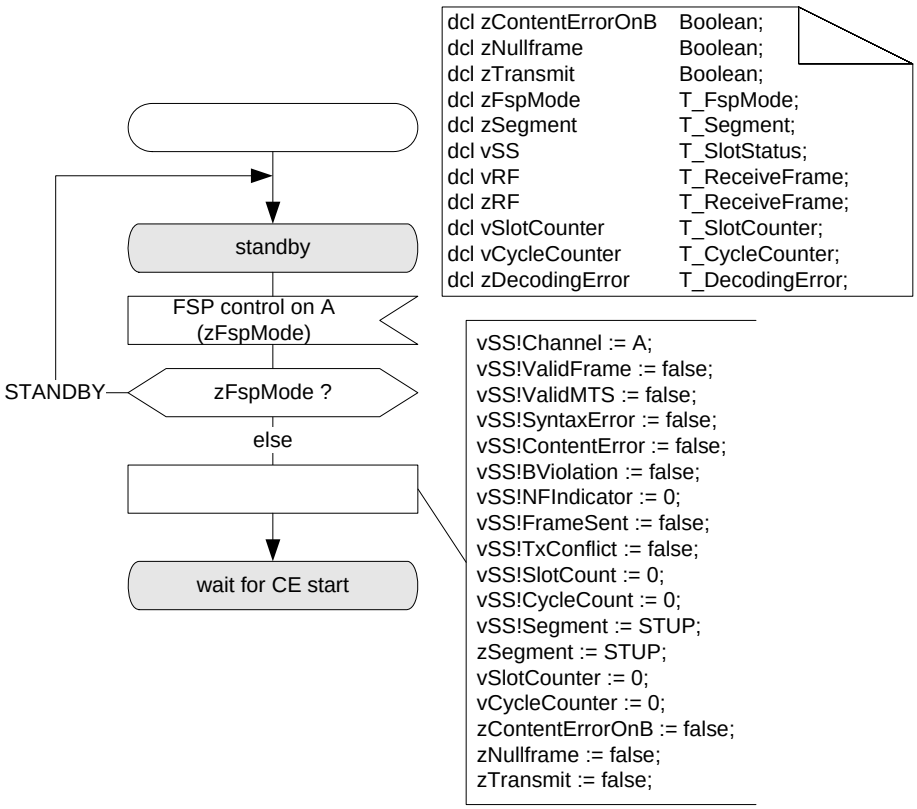

Figure 6-5: FSP process [FSP_A].

As depicted in Figure 6-6, a node shall enter the *FSP:standby* state from any state within the FSP process (with the exception of the *FSP:standby* state itself) if the protocol operation control process sets the FSP process to the STANDBY mode.

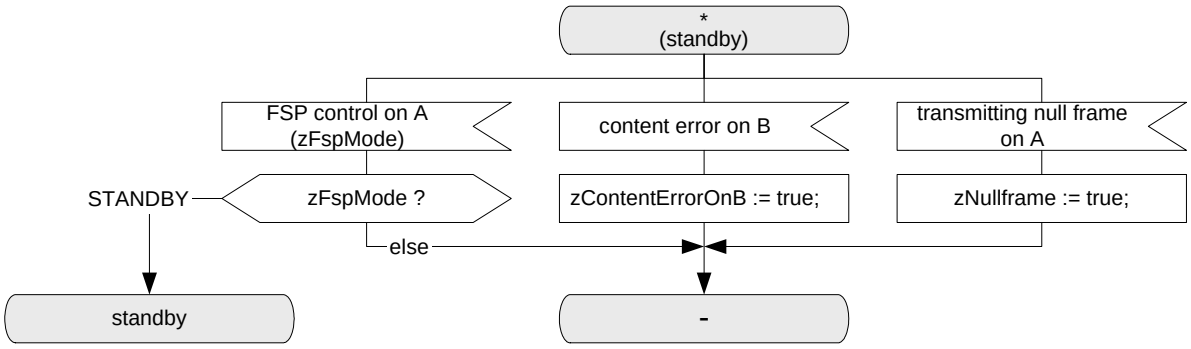

Figure 6-6: FSP control [FSP_A].

In addition, the node shall apply cross-channel content checks to identify cross-channel inconsistencies whenever *zSegment* = STATIC.[82]

As depicted in Figure 6-7, a node shall terminate the FSP process upon occurrence of the terminate event issued by the protocol operation control process.

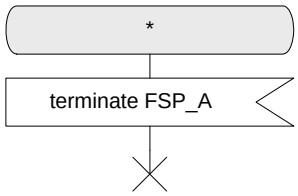

Figure 6-7: Termination of the FSP process [FSP_A].

Depending on the operation mode of the FSP process and the segment, the FSP process relays decoded wakeup patterns from the WUPDEC process to the CHI as shown in Figure 6-8.

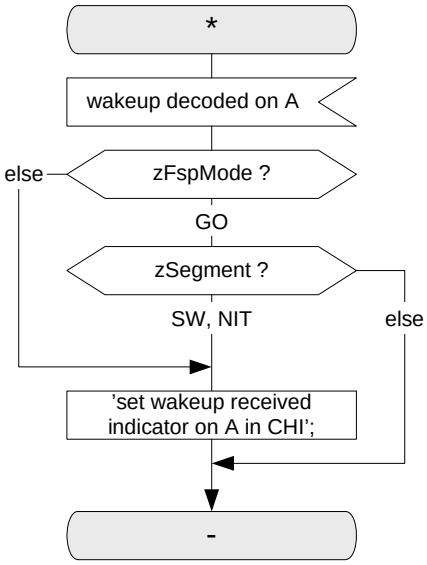

Figure 6-8: CHI update of a decoded wakeup pattern [FSP_A].

6.3.2 Macro SLOT_SEGMENT_END

The macro SLOT_SEGMENT_END that is depicted in Figure 6-9 shall be called within the FSP process

1. at the end of each static slot,
2. at the end of each dynamic slot if a dynamic segment is configured (i.e., *gNumberOfMinislots* > 0),
3. at the end of the symbol window if the symbol window is configured (i.e., *gdSymbolWindow* > 0), and
4. at the end of the network idle time.

If a valid frame was received, the sync frame indicator of the received frame is set, and no content error was detected on the other channel a node shall assert *valid sync frame on A (vRF)*. Such a frame is called a valid sync frame.

If the FSP process is in the GO mode a node shall make the slot status *vSS* and received frame data *vRF* available to the CHI.

[82] *zSegment* is STATIC during the static segment in normal operation, but can also be STATIC during some portions of startup.

A node shall initialize the slot status *vSS* for aggregation in the subsequent slot.

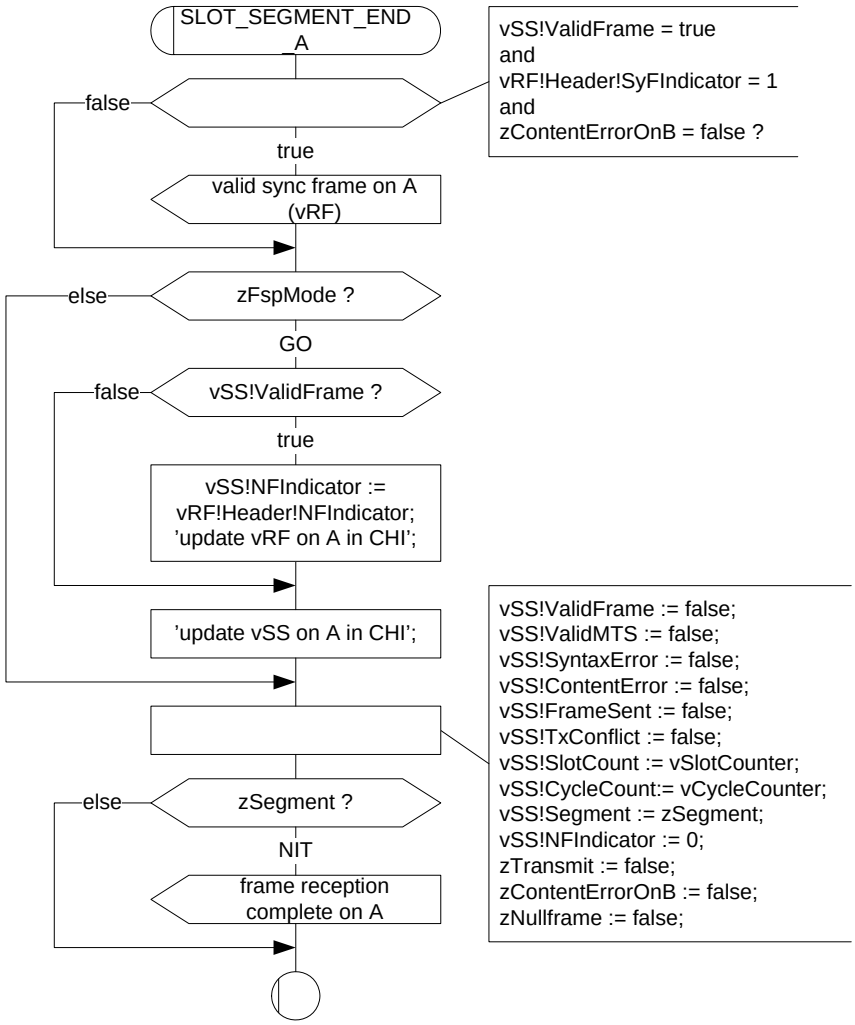

Figure 6-9: Slot and segment end macro [FSP_A].

6.3.3 *FSP:wait for CE start* state

The *FSP:wait for CE start* state and the transitions out of this state are depicted in Figure 6-10.

For each configured communication channel a node shall remain in the *FSP:wait for CE start* state until either

1. a communication element start is received, or
2. the node starts transmitting a communication element on the channel.

If either a slot boundary or one of the four segment boundaries is crossed then the node shall execute the SLOT_SEGMENT_END macro to provide the current slot status, and any frame data that may have been received, to the host interface for further processing. In this case the node shall remain in the *FSP:wait for CE start* state.

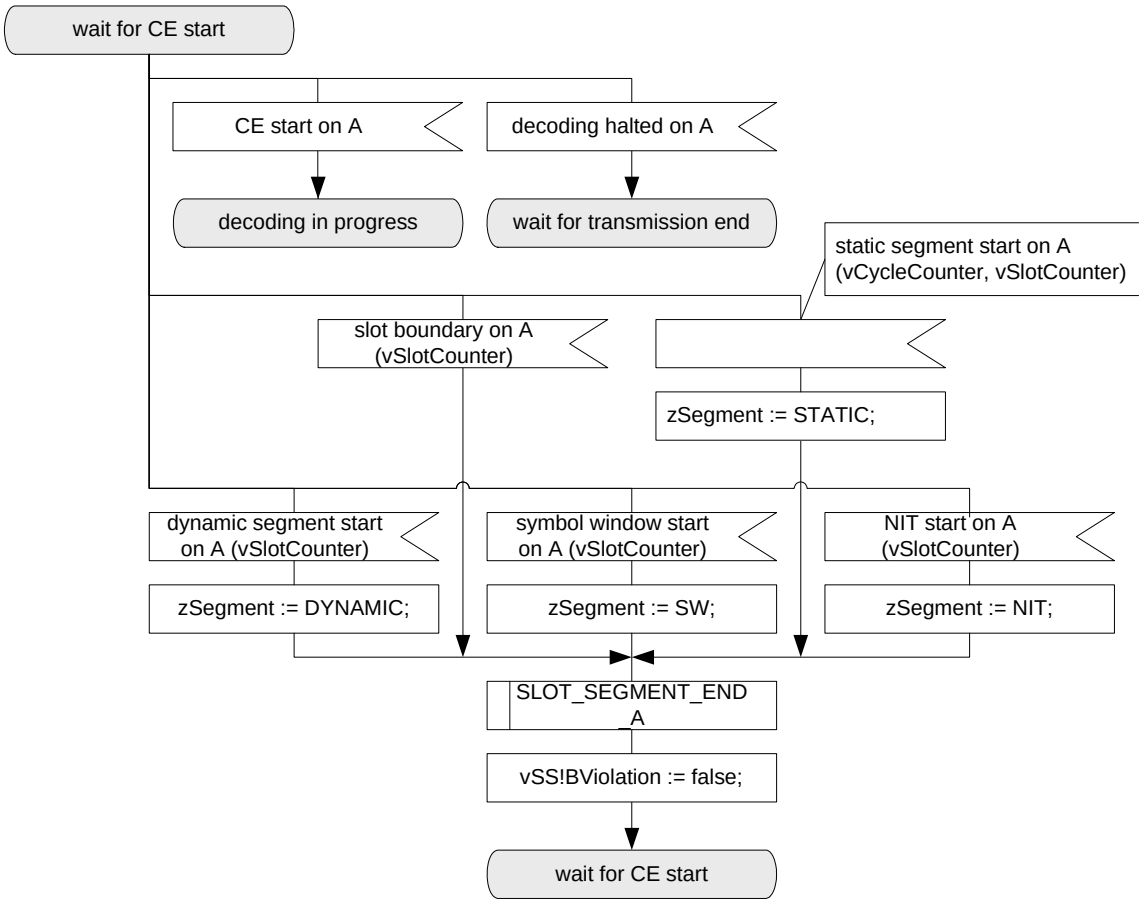

Figure 6-10: Transitions from the *FSP:wait for CE start* state [FSP_A].

6.3.4 *FSP:decoding in progress* state

The *FSP:decoding in progress* state and the transitions out of this state are depicted in Figure 6-11 and Figure 6-12.

For each configured communication channel a node shall remain in the *FSP:decoding in progress* state until either

1. the node starts transmitting on the communication channel, or
2. a decoding error occurs on the communication channel, or
3. a syntactically correct frame is decoded on the communication channel, or
4. a CAS/MTS symbol was decoded, or
5. a slot boundary or one of the four segment boundaries is crossed.

If either a slot boundary or one of the four segment boundaries is crossed then the node shall execute the SLOT_SEGMENT_END macro to provide the current slot status, and any frame data that may have been received, to the host interface for further processing.

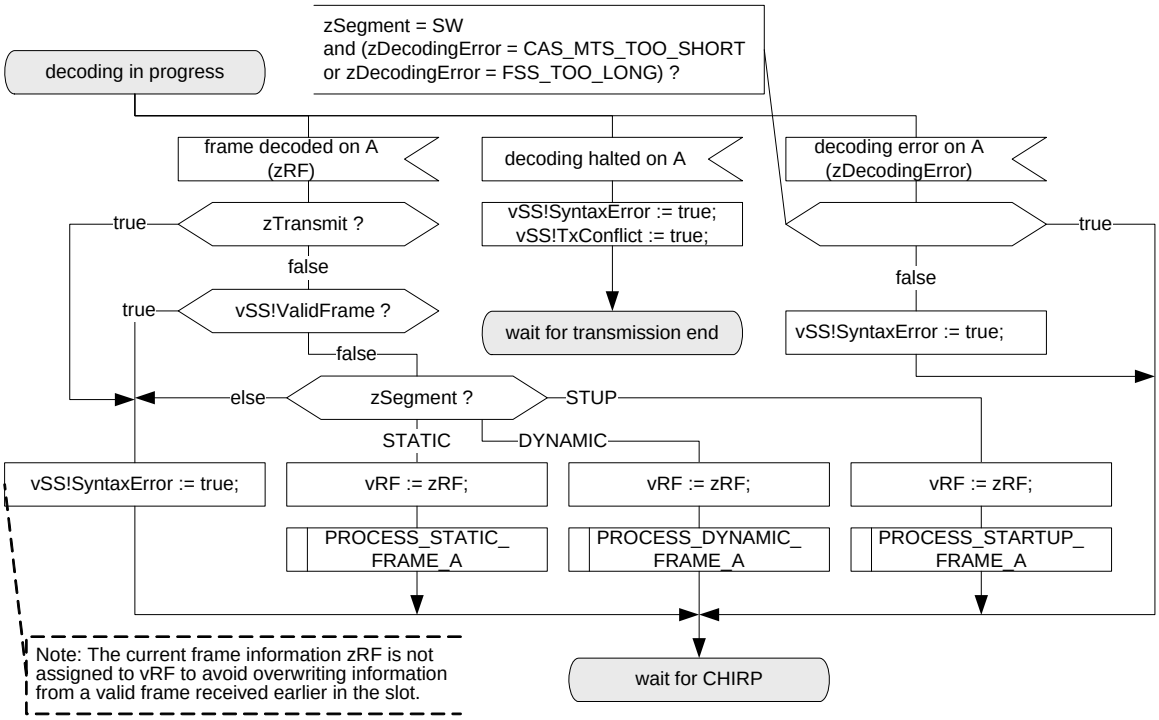

Figure 6-11: Transitions from the *FSP:decoding in progress* state [FSP_A].

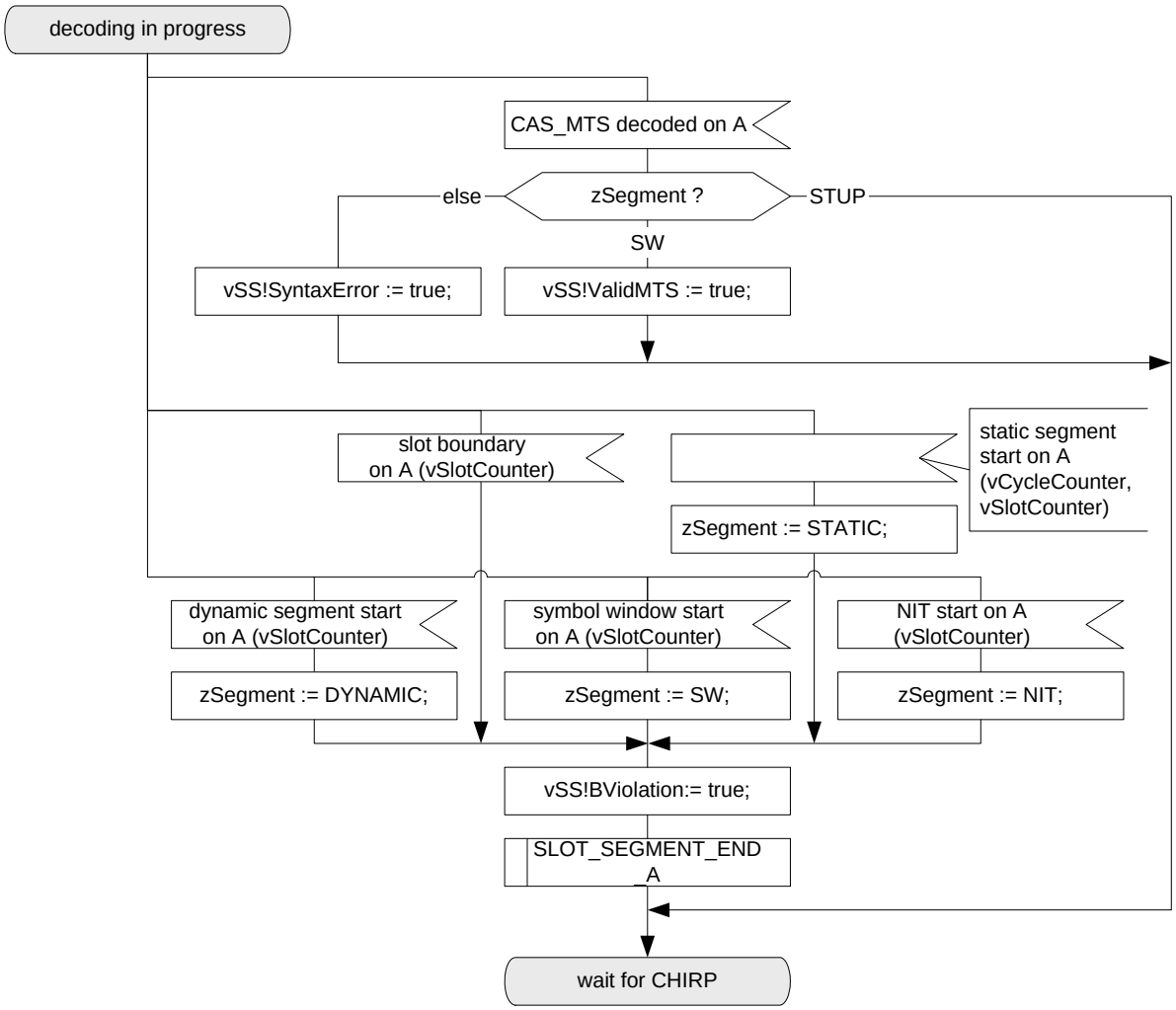

Figure 6-12: Transitions from the *FSP:decoding in progress* state [FSP_A].

6.3.4.1 Frame reception checks during non-synchronized operation

The frame acceptance checks that the node shall apply during non-synchronized operation are defined in the macro PROCESS_STARTUP_FRAME depicted in Figure 6-13.

For each configured communication channel the node shall accept each frame that fulfills all of the following criteria:

1. The frame ID included in the header of the frame is greater than 0 and not larger than the number of the last static slot *gNumberOfStaticSlots*.
2. The payload length included in the header of the frame equals the globally configured length for static frames *gPayloadLengthStatic*.
3. The sync frame indicator included in the header is set to one.
4. The startup frame indicator included in the header is set to one.
5. If the null frame indicator *vRF!Header!NFIndicator* is set to zero then the *vRF!Header!PPIndicator* is not set to one.
6. The cycle counter included in the header of the frame is not larger than the configured maximum cycle counter *gCycleCountMax*.

A frame that passes these checks is called a valid startup frame.

If the cycle count value included in the header of a valid startup frame is even then the frame is called a valid even startup frame.

If the cycle count value included in the header of a valid startup frame is odd then the frame is called a valid odd startup frame.

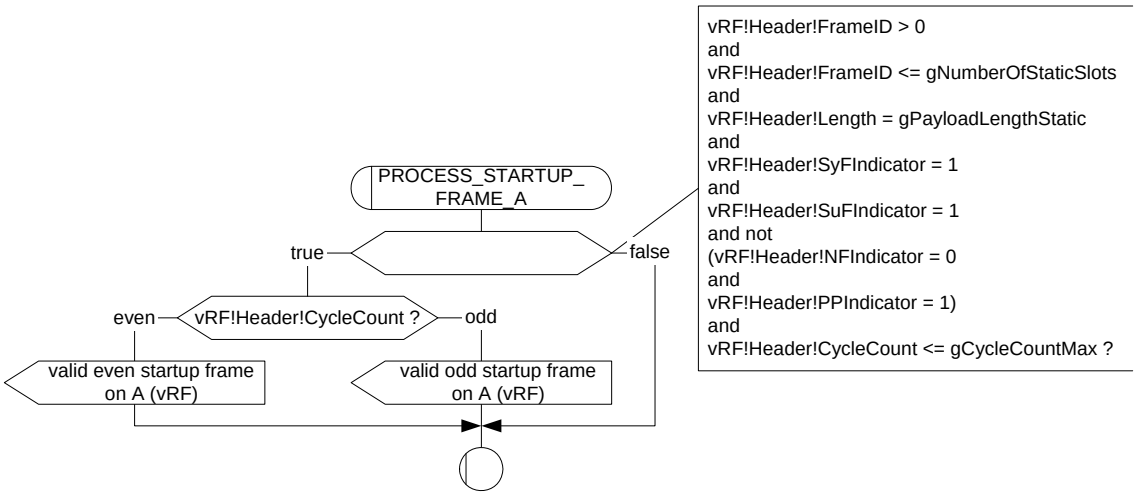

Figure 6-13: Frame acceptance checks during non-synchronized operation [FSP_A].

6.3.4.2 Frame reception checks during synchronized operation

6.3.4.2.1 Frame reception checks in the static segment

Figure 6-14 depicts the frame reception timing that must be met by a syntactically valid frame in the static segment.

The frame acceptance checks that the node shall apply during synchronized operation in the static segment are defined in the macro PROCESS_STATIC_FRAME depicted in Figure 6-15.

For each configured communication channel the node shall accept the first frame that fulfills all of the following criteria:

1. The frame is contained within one static slot.
2. The payload length included in the header of the frame matches the globally configured value of the payload length of a static frame held in *gPayloadLengthStatic*.
3. The frame ID included in the header of the frame equals the value of the slot counter *vSlotCounter*.
4. The cycle count included in the header of the frame matches the value of the cycle counter *vCycle-Counter*.
5. If the startup frame indicator in the header *vRF!Header!SuFIndicator* is set to one then the *vRF!Header!SyFIndicator* is not set to zero.
6. If the null frame indicator *vRF!Header!NFIndicator* is set to zero then the *vRF!Header!PPIndicator* is not set to one.

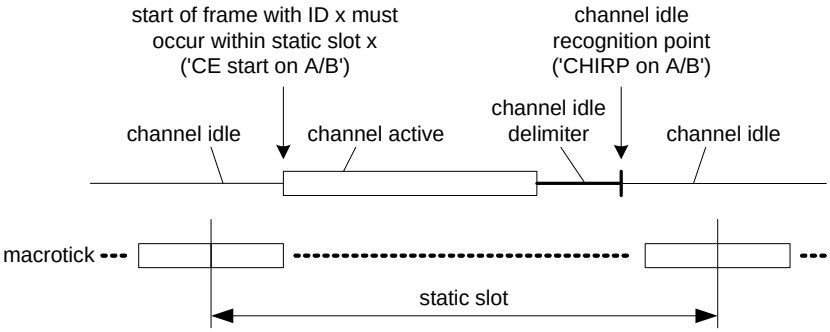

Figure 6-14: Frame reception timing for a static slot.

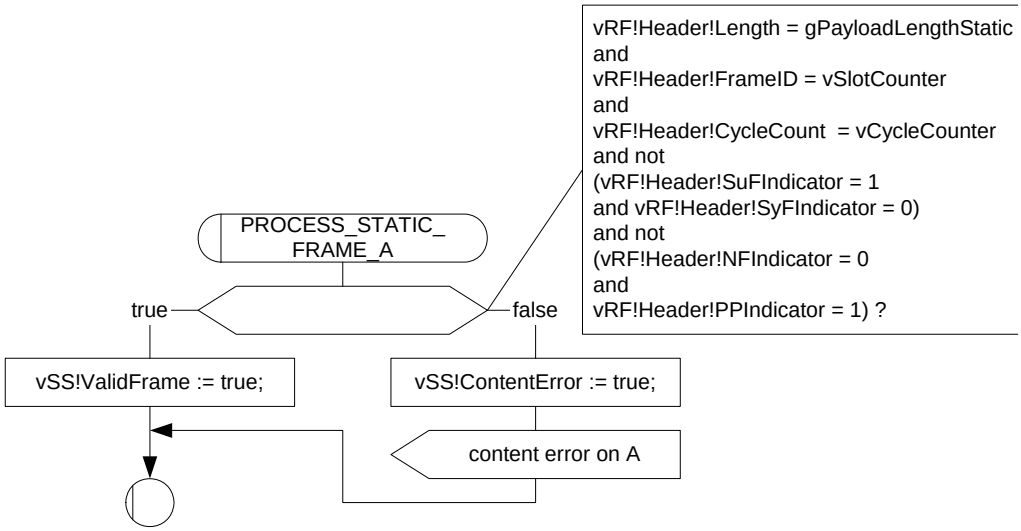

Figure 6-15: Frame acceptance checks for the static segment [FSP_A].

6.3.4.2.2 Frame reception checks in the dynamic segment

Figure 6-16 depicts the frame reception timing that must be met by a syntactically valid frame in the dynamic segment.

The frame acceptance checks that the node shall apply during synchronized operation in the dynamic segment are defined in the macro PROCESS_DYNAMIC_FRAME depicted in Figure 6-17.

For each configured communication channel the node shall accept the first frame that fulfills all of the following criteria:

1. The frame ID included in the header of the frame is greater than 0 and matches the value of the slot counter *vSlotCounter*.
2. The cycle count included in the header of the frame matches the value of the cycle counter *vCycle-Counter*.
3. The sync frame indicator included in the header is set to zero.
4. The startup frame indicator included in the header is set to zero.
5. The null frame indicator included in the header is set to one.

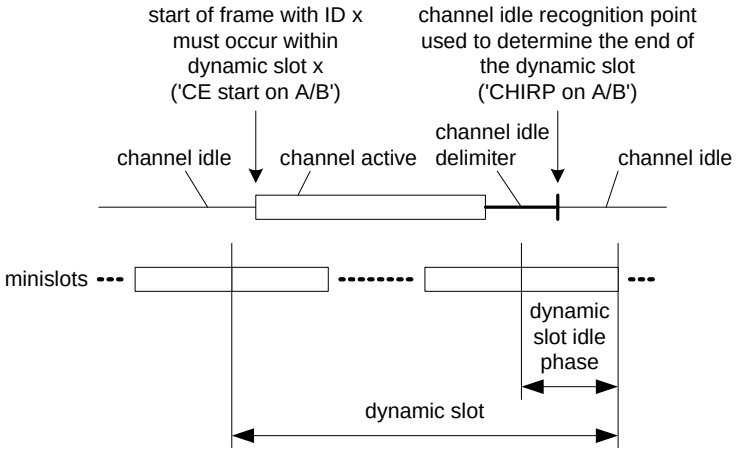

Figure 6-16: Frame reception timing for a dynamic slot.

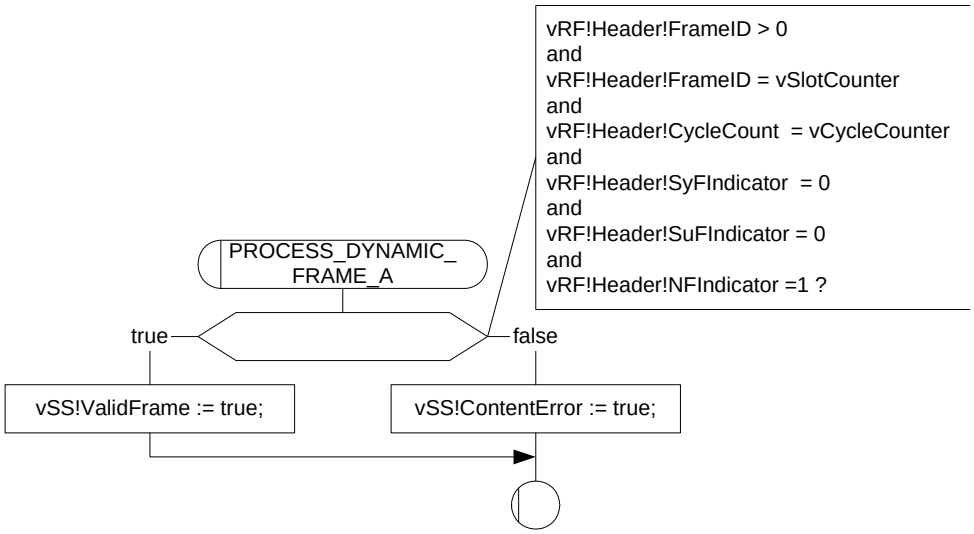

Figure 6-17: Frame acceptance checks for the dynamic segment [FSP_A].

6.3.5 *FSP:wait for CHIRP* state

The *FSP:wait for CHIRP* state and the transitions out of this state are depicted in Figure 6-18.

For each configured communication channel a node shall remain in the *FSP:wait for CHIRP* state until either

1. the channel idle recognition point is identified on the communication channel, or
2. the node starts transmitting on the communication channel.

If either a slot boundary or one of the four segment boundaries is crossed then the node shall execute the SLOT_SEGMENT_END macro to provide the current slot status, and any frame data that may have been received, to the host interface for further processing. In this case the node shall remain in the *FSP:wait for CHIRP* state.

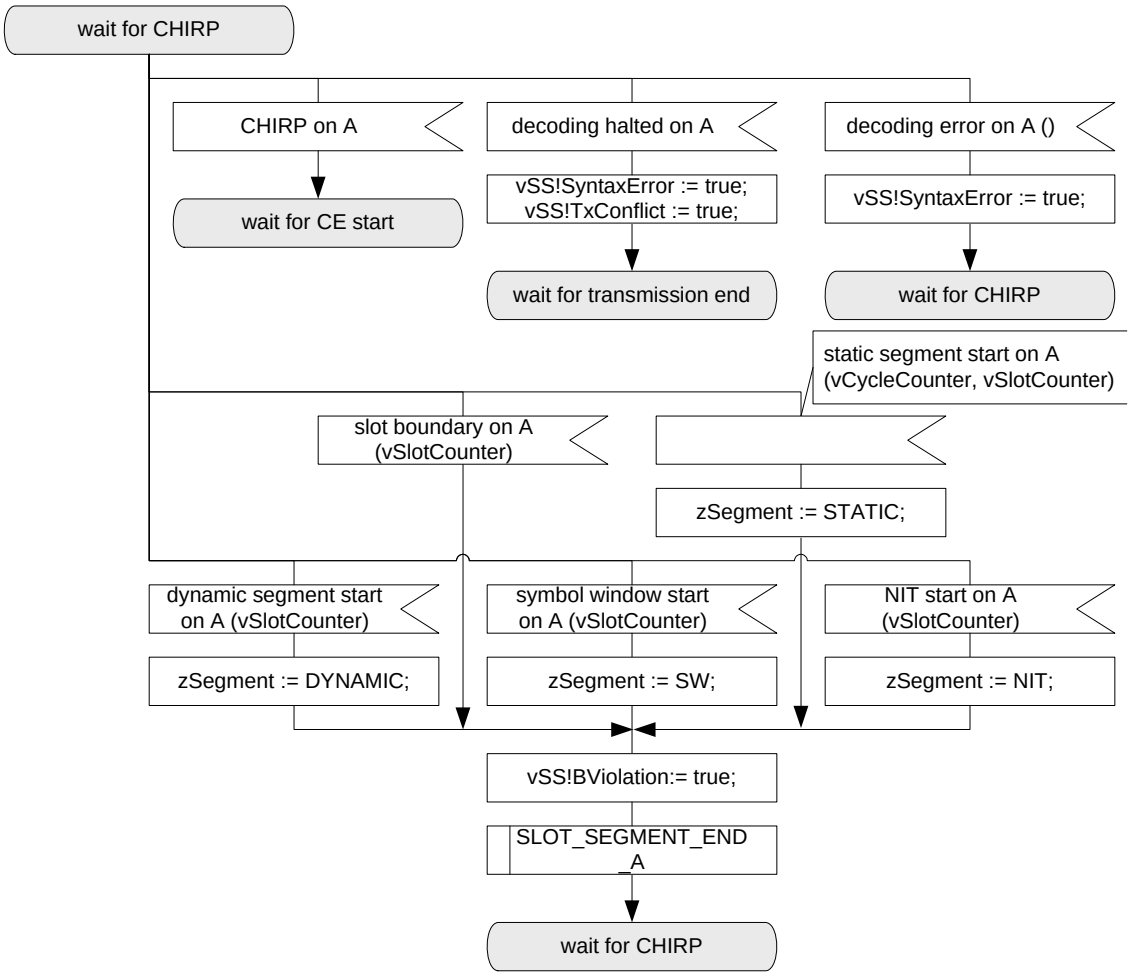

Figure 6-18: Transitions from the *FSP:wait for CHIRP* state [FSP_A].

6.3.6 *FSP:wait for transmission end* state

The *FSP:wait for transmission end* state and the transitions out of this state are depicted in Figure 6-19.

For each configured communication channel a node shall remain in the *FSP:wait for transmission end* state until either

1. the transmission ends on the channel, or
2. the slot boundary or one of the four segment boundaries is crossed.

If either a slot boundary or one of the four segment boundaries is crossed then the node shall signal a fatal protocol error to the protocol operation control process.

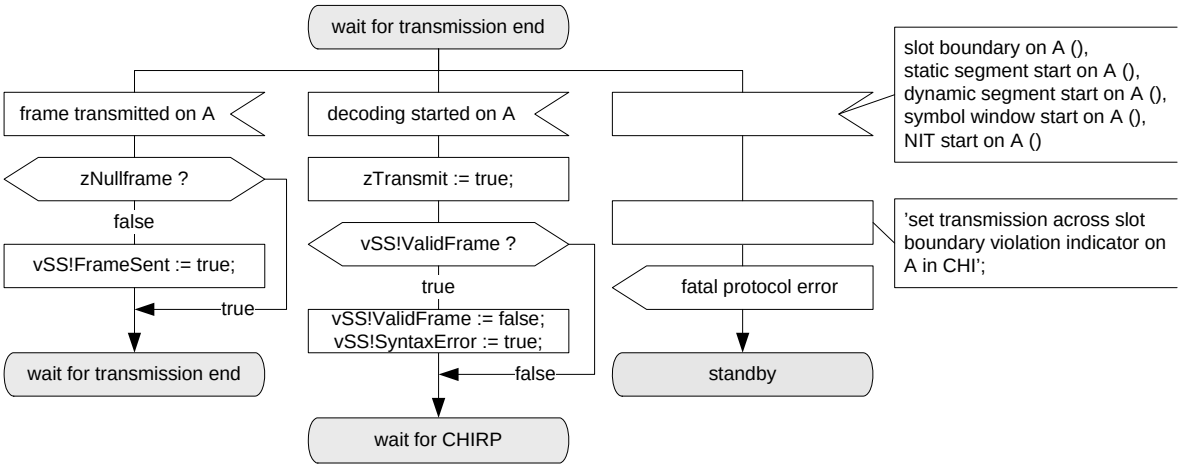

Figure 6-19: Transitions from the *FSP:wait for transmission end* state [FSP_A].

Chapter 7
Wakeup and Startup

This chapter describes the protocol mechanisms available to allow a node to cause a transition of a FlexRay cluster from a sleep mode to a mode where nodes are ready to begin synchronized operation (wakeup) and to allow a node to either initiate synchronized operation or integrate into a cluster that is already operating (startup).

First the protocol aspects of cluster wakeup are described in detail. Additional material in the form of application notes related to the interaction between the communication controller and host, as well as describing techniques for wakeup during operation, may be found in Appendix C.

Following the wakeup section, communication startup is described. This section also describes the integration (or reintegration) of nodes into a communication cluster.

7.1 Cluster wakeup

This section describes the procedure[83] used by communication controllers to initiate the remote cluster wakeup. The following procedures assume that the bus drivers in the system support the optional remote wakeup detection capability.

7.1.1 Principles

The minimum prerequisite for a cluster wakeup is that the receivers of all bus drivers be supplied with power. A bus driver has the ability to wake up the other components of its node when it receives a wakeup pattern on its channel. At least one node in the cluster needs an external wakeup source.

The host completely controls the wakeup procedure[84]. The communication controller provides the host the ability to transmit a special wakeup pattern (see Chapter 3) on each of its available channels separately. The wakeup pattern must not be transmitted on both channels at the same time. This is done to prevent a faulty node from disturbing communication on both channels simultaneously with the transmission. The host must configure which channel the communication controller shall wake up. The communication controller ensures that ongoing communication on this channel is not disturbed.

The wakeup pattern then causes any fault-free receiving node to wake up if it is still asleep. Generally, the bus driver of the receiving node recognizes the wakeup pattern and triggers the node wakeup. The communication controller needs to recognize the wakeup pattern only during the wakeup (for collision resolution) and startup phases.

The communication controller cannot verify whether all nodes connected to the configured channel are awake after the transmission of the wakeup pattern[85] since these nodes cannot give feedback until the startup phase. The host shall be aware of possible failures of the wakeup and act accordingly.

---

[83] To simplify discussion, the sequence of tasks executed while triggering the cluster wakeup is referred to here as the wakeup "procedure" even though it is realized as an SDL macro, and not an SDL procedure. The normal grammatical use of the term is intended rather than the precise SDL definition. Since SDL processes are not used in the wakeup mechanism, the usage does not introduce ambiguity.

[84] The host may force a mode change from wakeup mode to the *POC:ready* state. Note, however, that a forced mode-change to the *POC:ready* state during wakeup may have consequences regarding the consistency of the cluster.

[85] For example, the transmission unit of the bus driver may be faulty.

The wakeup procedure supports the ability for single-channel devices in a dual-channel system to initiate cluster wakeup by transmitting the wakeup pattern on the single channel to which they are connected. Another node, which has access to both channels, then assumes the responsibility for waking up the other channel and transmits a wakeup pattern on it (see Appendix C).

The wakeup procedure tolerates any number of nodes simultaneously trying to wake up a channel and resolves this situation such that eventually only one node transmits the wakeup pattern. Additionally, the wakeup pattern is collision resilient; so even in the presence of a fault causing two nodes to simultaneously transmit a wakeup pattern the signal resulting from the collision can still wake up the other nodes.

7.1.2 Description

The wakeup procedure is a subset of the Protocol Operation Control (POC) process. The relationship between the POC and the other protocol processes is depicted in Figure 7-1[86].

[86] The dark lines represent data flows between mechanisms that are relevant to this chapter. The lighter gray lines are relevant to the protocol, but not to this chapter.

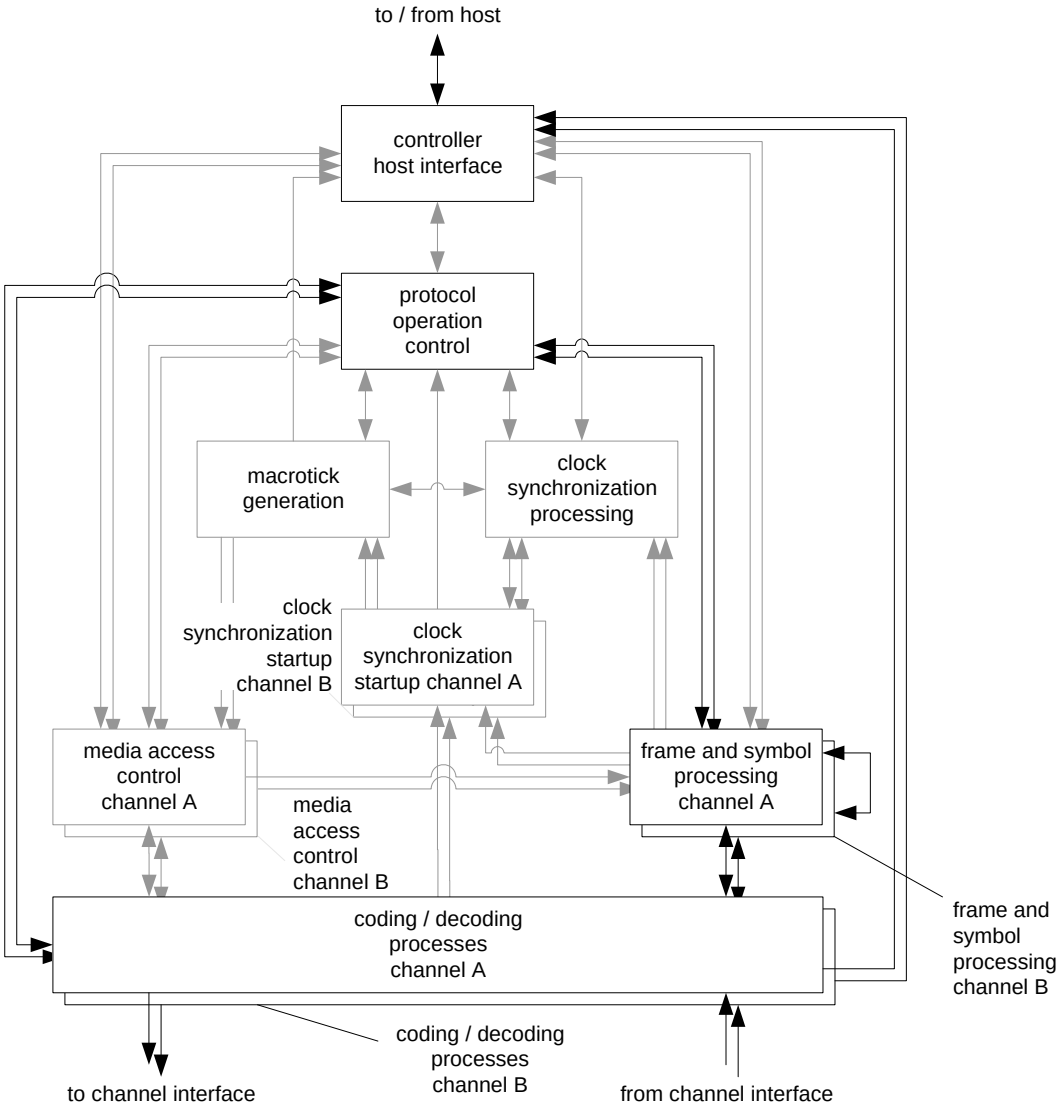

Figure 7-1: Protocol operation control context.

7.1.3 Wakeup support by the communication controller

The host must initialize the wakeup of the FlexRay cluster. The host has to configure the wakeup channel *pWakeupChannel* while the communication controller is in the *POC:config* state.

The host commands its communication controller to send a wakeup pattern on channel *pWakeupChannel* while the communication controller is in the *POC:ready* state. The communication controller then leaves the *POC:ready* state, begins the wakeup procedure (see Figure 7-2) and tries to transmit a wakeup pattern on the configured channel. Upon completion of the procedure it signals back the status of the wakeup attempt to the host (see section 9.3.1.3.1).

The host must properly configure the communication controller before it may trigger the cluster wakeup.

In the SDL description the wakeup procedure is realized as a macro that is called by the protocol operation control state machine (see Chapter 2).

7.1.3.1 Wakeup state diagram

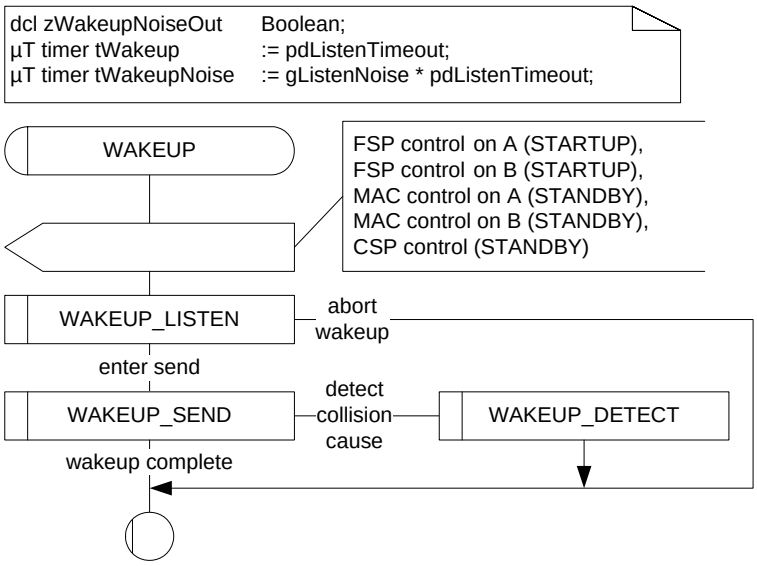

Figure 7-2: Structure of the wakeup state machine [POC].

The parameter *pWakeupChannel* identifies the channel that the communication controller is configured to wake up. The host can only configure the wakeup channel in the *POC:config* state. After the communication controller has entered the *POC:ready* state the host can initiate wakeup on channel *pWakeupChannel*.

Upon completing the wakeup procedure the communication controller shall return into the *POC:ready* state and signal to the host the result of the wakeup attempt.

The return condition of the WAKEUP macro is formally defined as *T_WakeupStatus* in section 2.2.1.3 in Definition 2-5.

The return status variable *vPOC!WakeupStatus* is set by the POC to

- UNDEFINED, if the host has not issued a WAKEUP command since the last entry to the *POC:default config* state (see Figure 2-11), or when the POC has not completed[87] a wakeup requested by the host,
- RECEIVED_HEADER, if the communication controller has received a frame header without coding violation on either channel during the initial listen phase,
- RECEIVED_WUP, if the communication controller has received a valid wakeup pattern on channel *pWakeupChannel* during the initial listen phase,
- COLLISION_HEADER, if the communication controller has detected a collision during wakeup pattern transmission by receiving a valid header during the ensuing detection phase,
- COLLISION_WUP, if the communication controller has detected a collision during wakeup pattern transmission by receiving a valid wakeup pattern during the ensuing detection phase,
- COLLISION_UNKNOWN, if the communication controller has detected a collision but did not detect a subsequent reception event that would allow the collision to be categorized as either COLLISION_HEADER or COLLISION_WUP,
- TRANSMITTED, if the wakeup pattern was completely transmitted.

[87] This could occur if the wakeup is still in progress when the variable is examined by the host, if the POC aborted the wakeup prior to completion due to an IMMEDIATE_READY, DEFERRED_READY, DEFFERED_HALT, or FREEZE command from the host, or if the POC did not attempt a wakeup because *pWakeupPattern* < 2.

7.1.3.2 The *POC:wakeup listen* state

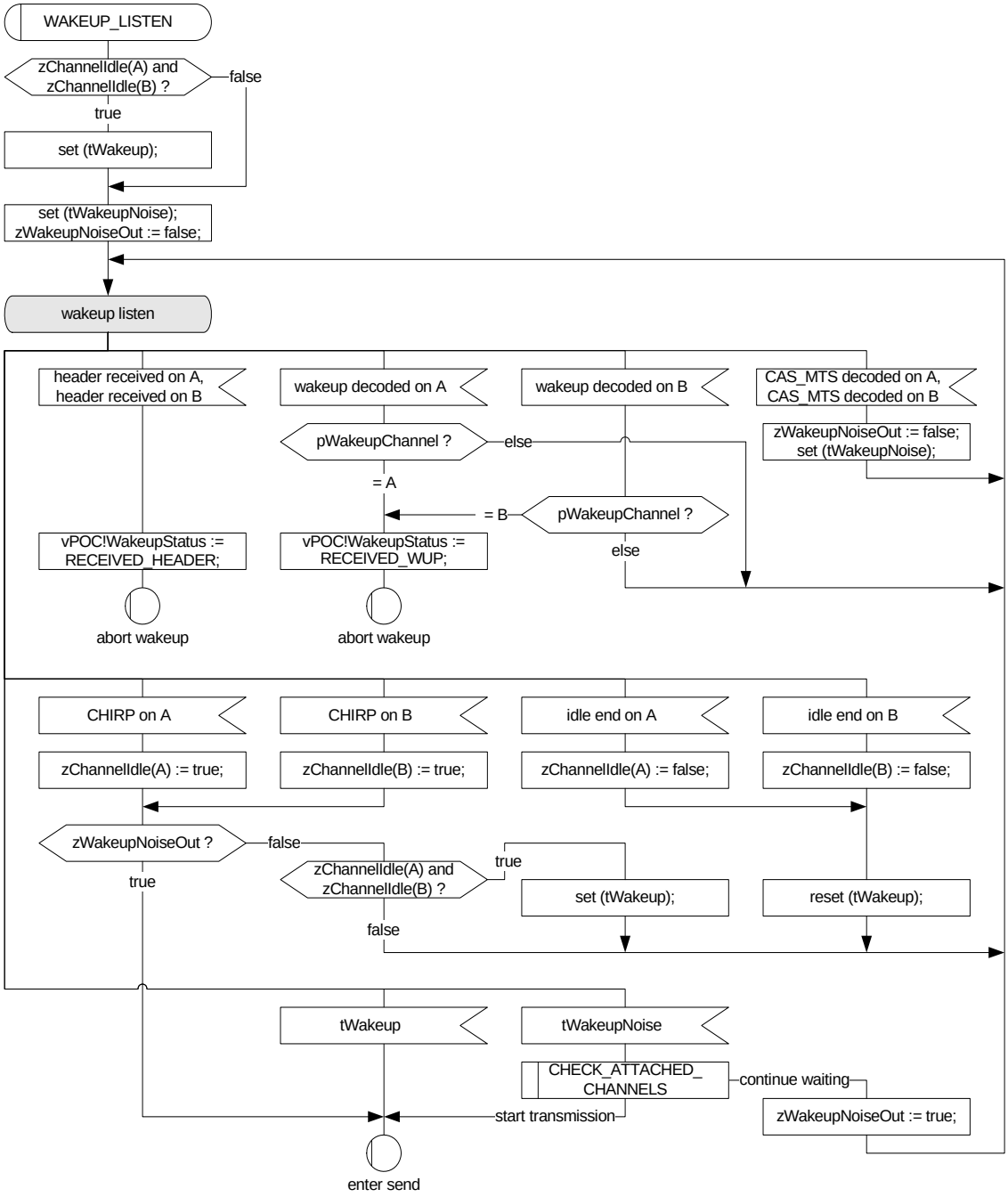

Figure 7-3: Transitions from the *POC:wakeup listen* state [POC].[88]

The purpose of the *POC:wakeup listen* state is to inhibit the transmission of the wakeup pattern if existing communication or a startup is already in progress.

[88] Note that if all attached channels are stuck continuously active low POC will remain in the wakeup until the host commands it to a different state.

The timer *tWakeup* enables a fast cluster wakeup in a noise free environment, while the timer *tWakeup-Noise* enables wakeup under more difficult conditions when noise interference is present or if a single channel is permanently busy.

When ongoing communication is detected or a wakeup of *pWakeupChannel* is already in progress, the wakeup attempt is aborted.

7.1.3.3 The *POC:wakeup send* state

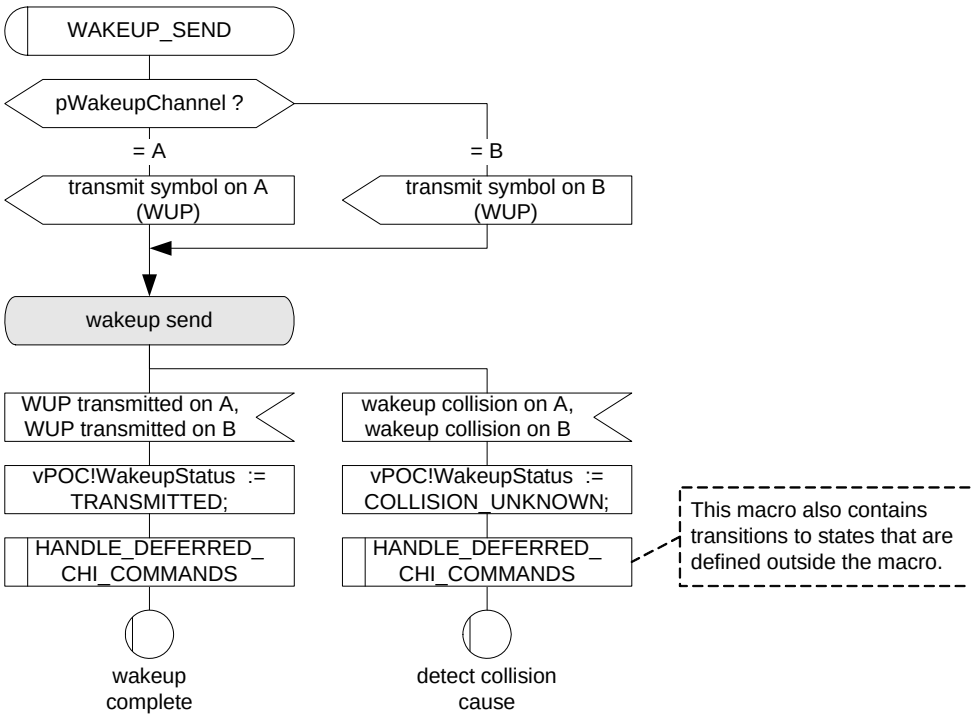

Figure 7-4: Transitions from the *POC:wakeup send* state [POC].

In this state, the communication controller transmits the wakeup pattern on the configured channel and checks for collisions.

Since the communication controller transmits the wakeup pattern on *pWakeupChannel*, it cannot really determine whether another node sends a wakeup pattern or frame on this channel during its transmission. Only during the idle portions of the wakeup pattern can it listen to the channel. If during one of these idle portions activity is detected, the communication controller leaves the send phase and enters a succeeding monitoring phase (*POC:wakeup detect* state) so that the cause of the collision might be identified and presented to the host.

7.1.3.4 The *POC:wakeup detect* state

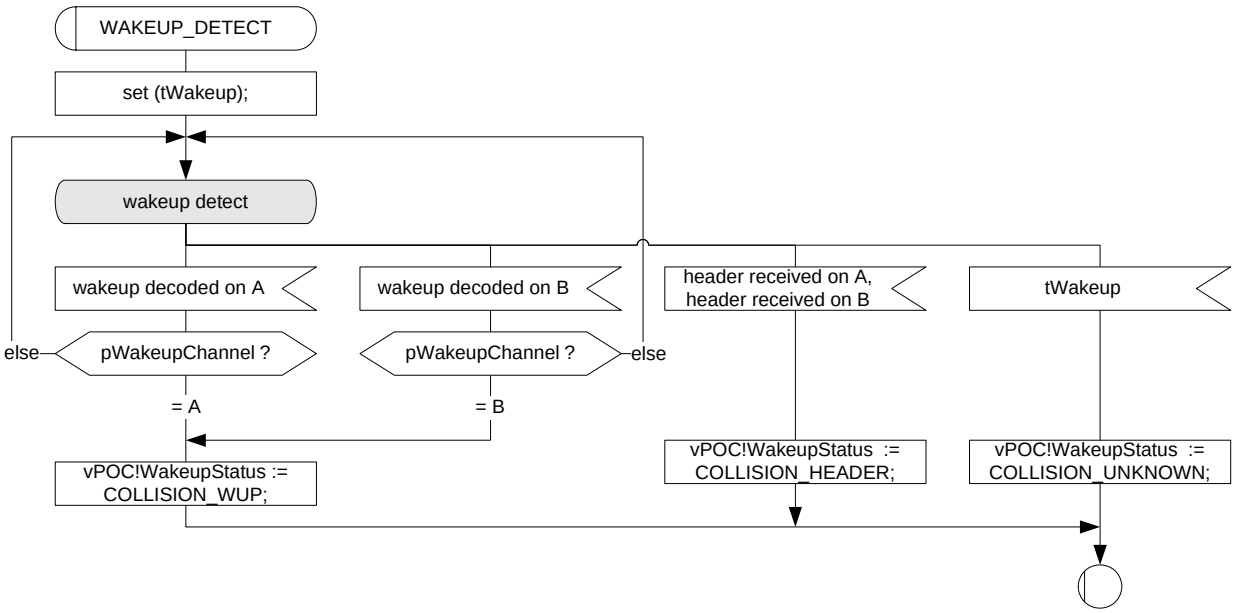

Figure 7-5: Transitions from the *POC:wakeup detect* state [POC].

In this state, the communication controller attempts to discover the reason for the wakeup collision encountered in the previous state (*POC:wakeup send*).

Note there are circumstances where a collision is detected between a WUS and a frame, but the mechanism is not able to identify the actual cause of the collision, returning COLLISION_UNKNOWN instead. If the node that transmitted the frame that collided with the WUS transmits another frame within approximately two cycles the identification procedure will return COLLISION_HEADER, but if the node does not continue transmission the result may still be COLLISION_UNKNOWN. As a result, it is not possible to rely on the existence of a COLLISION_HEADER indication if the cause of a collision was actually a collision with a frame. If COLLISION_HEADER is returned, however, it is certain that frames were present on the network.

This monitoring is bounded by the expiration of the timer *tWakeup*. The detection of either a wakeup pattern indicating a wakeup attempt by another node or the reception of a frame header indicating existing communication causes a direct transition to the *POC:ready* state.

7.2 Communication startup and reintegration

A TDMA based communication scheme requires synchrony and alignment of all nodes that participate in cluster communication. A fault-tolerant, distributed startup strategy is specified for initial synchronization of all nodes. This strategy is described in the following subsections.

7.2.1 Principles

7.2.1.1 Definition and properties

Before communication startup can be performed, the cluster has to be awake, so the wakeup procedure has to be completed before startup can commence. The startup is performed on all channels synchronously.

The action of initiating a startup process is called a coldstart. Only a limited number of nodes may initiate a startup, they are called the coldstart nodes. The process outlined below effectively describes the startup procedure of FlexRay clusters using any synchronization method. As the TT-E synchronization method uses a drastically simplified startup process, much of the description is not relevant for TT-E coldstart nodes as indicated in the textual and SDL description.

With the exception of TT-E clusters a coldstart attempt begins with the transmission of a collision avoidance symbol (CAS). Only the coldstart node that transmits the CAS transmits frames in the first four cycles after the CAS. It is then joined first by the other coldstart nodes (if present) and afterwards by all other nodes.

A coldstart node has the configuration parameter *pKeySlotUsedForStartup* set to true and is configured with a frame header containing a startup frame indicator set to one (see Chapter 4). At least two fault-free coldstart nodes are necessary for the cluster to start up. In TT-D clusters consisting of two nodes, each node shall be a coldstart node. Each startup frame shall also be a sync frame; therefore each coldstart node will also be a sync node. In order to accommodate the possibility of the failure of a coldstart node, a TT-D cluster consisting of three or more nodes should configure at least three nodes as coldstart nodes.[89]

A coldstart node that actively starts the cluster is also called a leading coldstart node. A coldstart node that integrates upon another coldstart node is also called a following coldstart node. By definition a TT-L coldstart node can never be a following coldstart node, as it is the sole coldstart node of its cluster. As TT-E coldstart nodes essentially skip the usual startup process, the distinction is meaningless for them.

A node is in "startup" if its protocol operation control machine is in one of the states contained in the STARTUP macro (*vPOC!State* is set to STARTUP during this time, see Figure 2-12). During startup, a node may only transmit startup frames. Any coldstart node shall wake up the cluster or determine that it is already awake before entering startup (see section C.1).

7.2.1.2 Principle of operation

The system startup consists of two logical steps. In the first step dedicated coldstart nodes start up. In the second step the other nodes integrate to the coldstart nodes.

7.2.1.2.1 Startup performed by the coldstart nodes

- Only the coldstart nodes can initiate the cluster start up.
- When the TT-D synchronization method is used by the cluster, each of the TT-D coldstart nodes finishes its startup as soon as stable communication with one of the other coldstart nodes is established. A TT-L coldstart node always finishes the startup after six cycles. A TT-E coldstart node finishes its startup as soon as it receives the first cycle start from its time gateway source node after having previously received information about rate and offset correction terms and confirmation that the time gateway source node is in *POC:normal active*.

7.2.1.2.2 Integration of the non-coldstart nodes

- A non-coldstart node requires at least two startup frames with different frame IDs for integration.In a TT-D cluster this condition ensures that each non-coldstart node always joins the majority of the coldstart nodes[90].
- Integrating non-coldstart nodes may start their integration before coldstart nodes have finished their startup.
- Integrating non-coldstart nodes in a TT-D cluster will not finish their startup until at least two coldstart nodes have finished their startup.

[89] This does not imply any restrictions concerning which nodes may initiate a cluster wakeup.
[90] This is the case under the assumption that the cluster contains exactly the three recommended coldstart nodes.

7.2.2 Description

The startup procedure is a subset of the Protocol Operation Control (POC) process. The relationship between the POC and the other protocol processes is depicted in Figure 7-6.

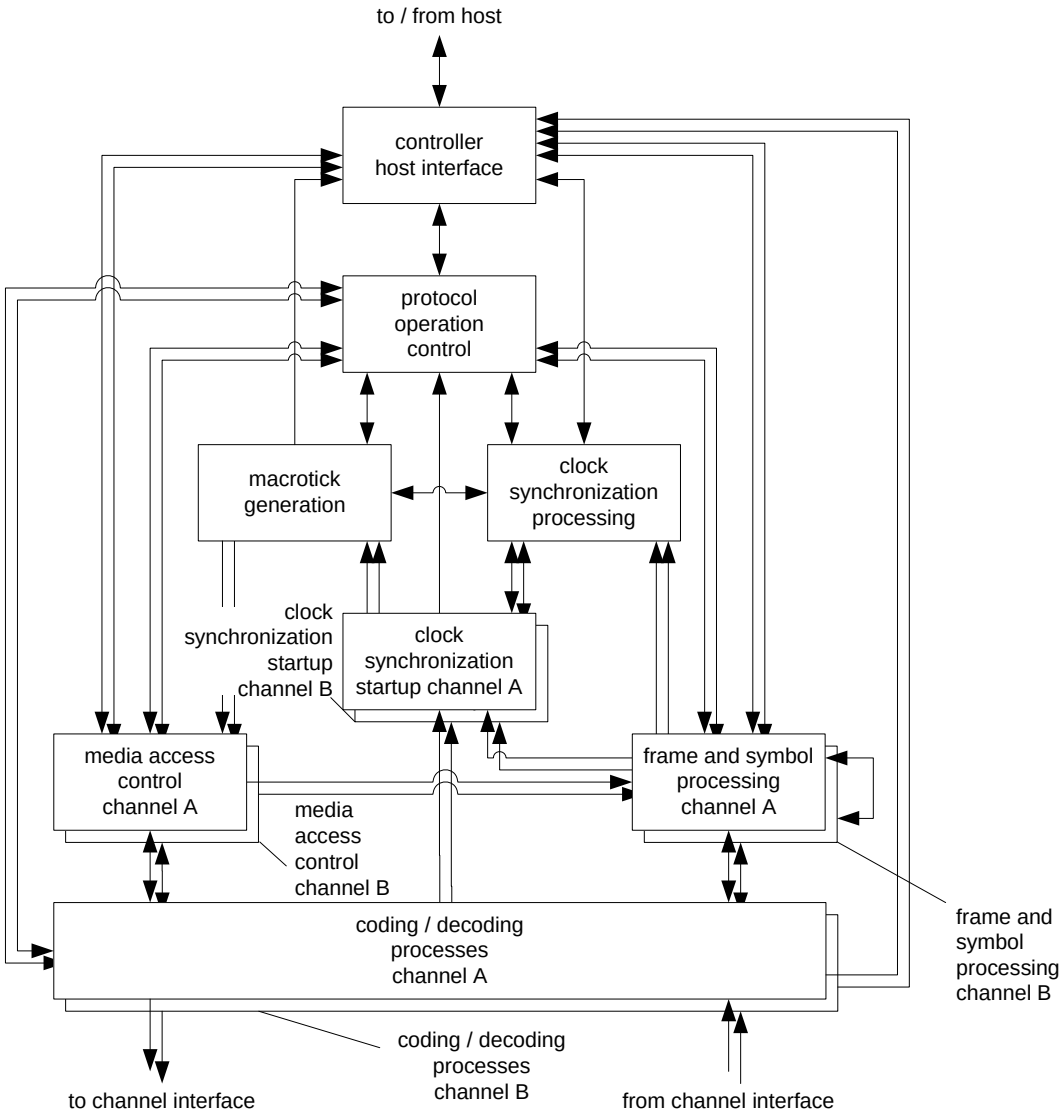

Figure 7-6: Protocol operation control context.

7.2.3 Coldstart inhibit mode

In coldstart inhibit mode the communication controller is prevented from initializing the TDMA communication schedule. The communication controller is automatically set into coldstart inhibit mode prior to entering the *POC:ready* state. The CC remains in this mode until the host commands the node to leave coldstart inhibit mode by means of an ALLOW_COLDSTART command issued after the POC enters the *POC:ready* state[91].

[91] Note there is no mechanism that allows the host to explicitly place the node into coldstart inhibit mode - this happens automatically as part of the process of reaching *POC:ready*.

While in coldstart inhibit mode, the communication controller is not allowed to assume the role of a leading coldstart node, i.e., entering the coldstart path is prohibited. The node is still allowed to integrate into a running cluster or to act as a following coldstart node, transmitting startup frames after another coldstart node has started the initialization of cluster communication. Once the node is synchronized and integrated into cluster communication, the coldstart inhibit mode does not restrict the node's ability to transmit frames.

The coldstart inhibit status of a node is represented by the *vColdstartInhibit* variable, with a value of true indicating the node is in coldstart inhibit mode and a value of false indicating the node is not in coldstart inhibit mode.

The coldstart inhibit mode can be used either to completely prohibit active startup attempts of a node (if the host never issues an ALLOW_COLDSTART command), or only delay them (if the host issues the CHI command during a startup procedure). As a result, the coldstart inhibit mode may be used to attempt to ensure that all fault-free coldstart nodes are ready for startup and in the *POC:coldstart listen* state before one of them initiates a coldstart attempt.

Note that a coldstart node in a TT-E cluster (i.e., a node with *pExternalSync* set to true) will never be in coldstart inhibit mode as it follows a fundamentally different path for startup. Such nodes will operate as a coldstart node even without the issuance of an ALLOW_COLDSTART command.

7.2.4 Startup state diagram

There are several ways that a node can enter communication. Section 7.2.4.1 describes the path followed by the leading coldstarter in a TT-D or TT-L cluster. Section 7.2.4.2 describes the paths available to a TT-D coldstart node that does not act as the leading coldstarter. Section 7.2.4.4 describes the path followed by a TT-E coldstart node. Section 7.2.4.5 describes the path of a non-coldstart node in all cluster types. All of these sections provide only an overview[92] of the operation - the precise behavior is defined by the SDL descriptions in the subsequent sections.

[92] Note that there are a large number of paths possible involving multiple executions of the STARTUP_PREPARE macro (for example, paths involving one or more aborted startup attempts). No overview of these paths is provided, but their behavior is explicitly defined by the SDL descriptions in this chapter.

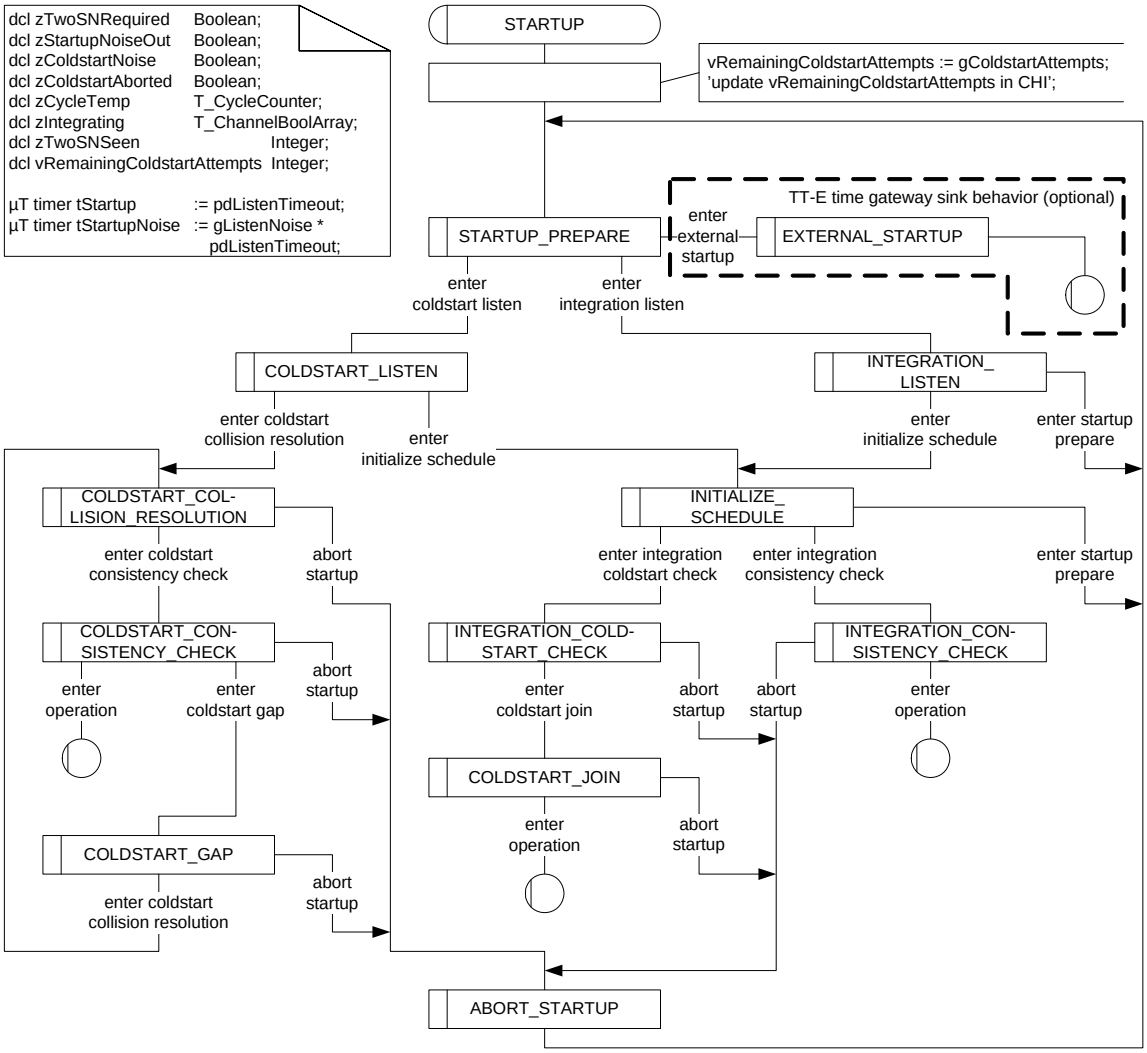

Figure 7-7: Startup state diagram [POC].

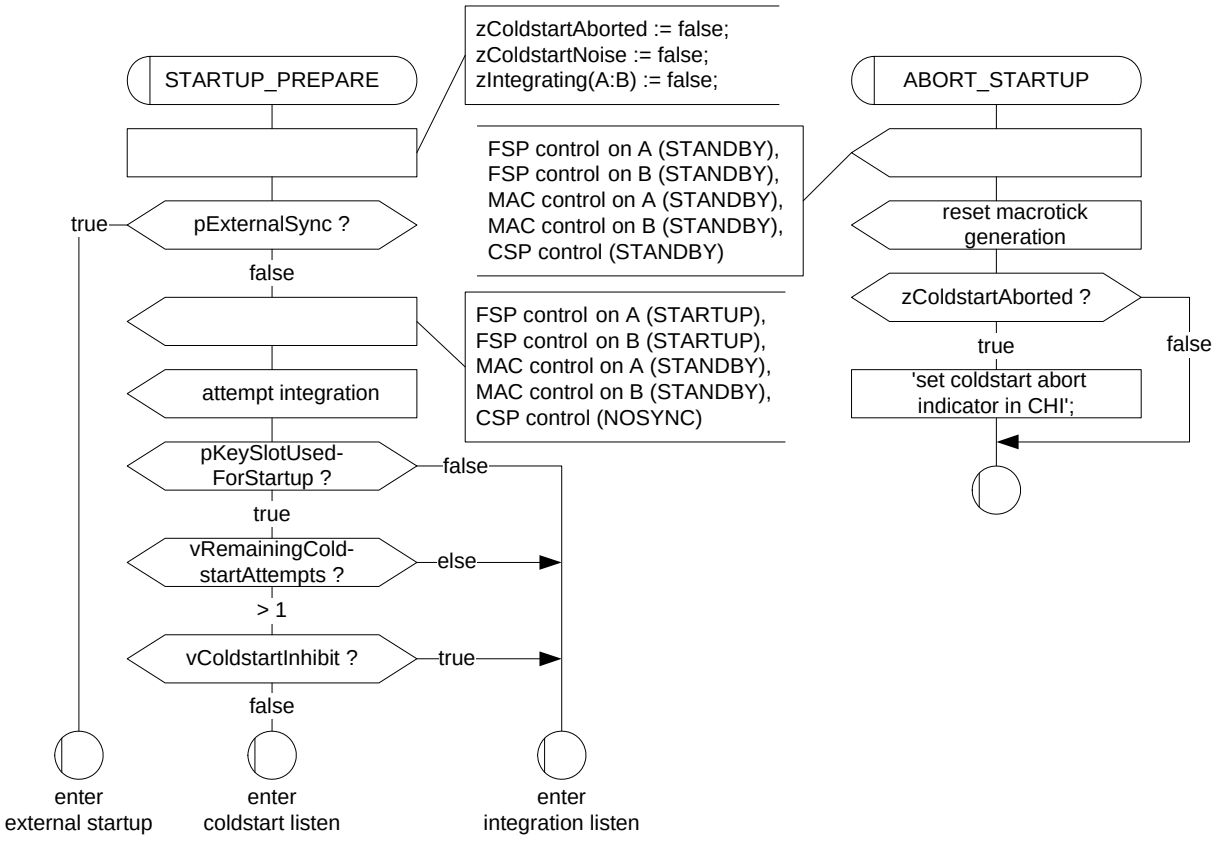

Figure 7-8: Helpful macros for startup [POC].

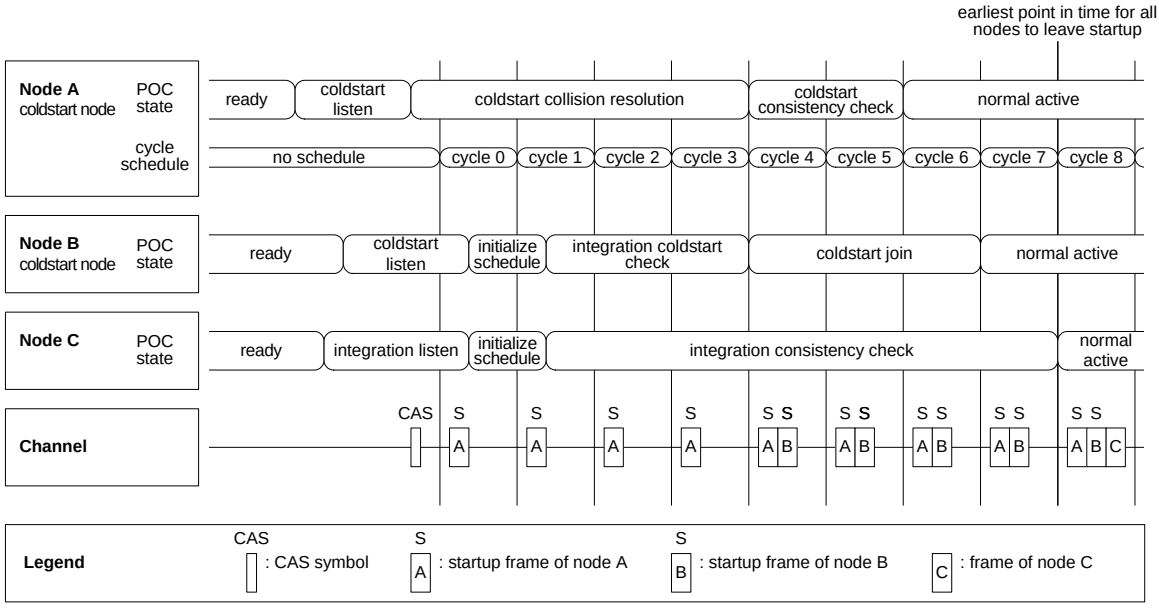

Figure 7-9: Example of state transitions for a fault-free startup[93].

7.2.4.1 Path of a TT-D leading coldstart node

Node A in Figure 7-9 follows this path and is therefore called a leading coldstart node.

When a coldstart node enters startup, it listens to its attached channels and attempts to receive FlexRay frames (see SDL macro COLDSTART_LISTEN in Figure 7-14).

If no communication[94] is received, the node commences a coldstart attempt. The initial transmission of a CAS symbol is succeeded by the first regular cycle. This cycle has the number zero.

From cycle zero on, the node transmits its startup frame (with the exception of the coldstart gap or the abort of the startup attempt). Since each coldstart node is allowed to perform a coldstart attempt, it may occur that several nodes simultaneously transmit the CAS symbol and enter the coldstart path. This situation is resolved during the first four cycles after CAS transmission. As soon as a node that initiates a coldstart attempt receives a CAS symbol or a frame header during these four cycles, it reenters the listen state. Consequently, only one node remains in this path (see SDL macro COLDSTART_COLLISION_RESOLUTION in Figure 7-16).

In cycle four, other coldstart nodes begin to transmit their startup frames. The node that initiated the coldstart collects all startup frames from cycle four and five and performs the clock correction as described in Chapter 8. If clock correction does not signal any errors and the node has received at least one valid startup frame pair, the node leaves startup and enters operation (see SDL macro COLDSTART_CONSISTENCY_CHECK in Figure 7-17).

7.2.4.2 Path of a TT-D following coldstart node

Node B in Figure 7-9 follows this path and is therefore called a following coldstart node.

When a coldstart node enters the startup, it listens to its attached channels and attempts to receive FlexRay frames (see SDL macro COLDSTART_LISTEN in Figure 7-14).

If communication[95] is received, it tries to integrate to a transmitting coldstart node[96]. It tries to receive a valid pair of startup frames to derive its schedule and clock correction from the coldstart node (see Chapter 8 and see SDL macro INITIALIZE_SCHEDULE in Figure 7-19).

If these frame receptions have been successful, it collects all sync frames and performs clock correction in the following double cycle. If clock correction does not signal any errors and if the node continues to receive sufficient frames from the same node it has integrated on, it begins to transmit its startup frame; otherwise it reenters the listen state (see SDL macro INTEGRATION_COLDSTART_CHECK in Figure 7-20).

If for the following three cycles the clock correction does not signal any errors and at least one other coldstart node is visible, the node leaves startup and enters operation. Thereby, it leaves startup at least one cycle after the node that initiated the coldstart (see SDL macro COLDSTART_JOIN in Figure 7-21).

Another path, not shown in Figure 7-9, is also possible for an integrating coldstart node. If, at the time of the execution of the STARTUP_PREPARE macro, the node is prevented from acting as a leading coldstarter (either because the *vColdstartInhibit* flag is set to true, or because the *vRemainingColdstartAttempts* variable indicates there are no remaining coldstart attempts) the node will instead begin to act as a normal integrating node, waiting for a leading coldstarter to begin transmissions that will initialize the schedule (see SDL macro INTEGRATION_LISTEN). Once such communication is detected, the node then executes the INITIALIZE_SCHEDULE macro and behaves as described earlier in this section.

[93] Please note that several simplifications have been made within this diagram to make it more accessible, e.g. the state transitions do not occur on cycle change, but well before that (see Chapter 8).
[94] See Figure 7-14 for exact definition.
[95] See Chapter 8 for exact definition.
[96] Presumably it is the node that initiated the coldstart, but not necessarily.

7.2.4.3 Path of a TT-L coldstart node

Node A in Figure 7-10 follows this path.

When a TT-L coldstart node enters startup, it listens to its attached channels and attempts to receive FlexRay frames (see SDL macro COLDSTART_LISTEN in Figure 7-14) even though it is the sole provider of startup frames of its cluster. As no communication can be received, the node soon commences a coldstart attempt. The initial transmission of a CAS symbol is succeeded by the first regular cycle, which is given a cycle number of zero.

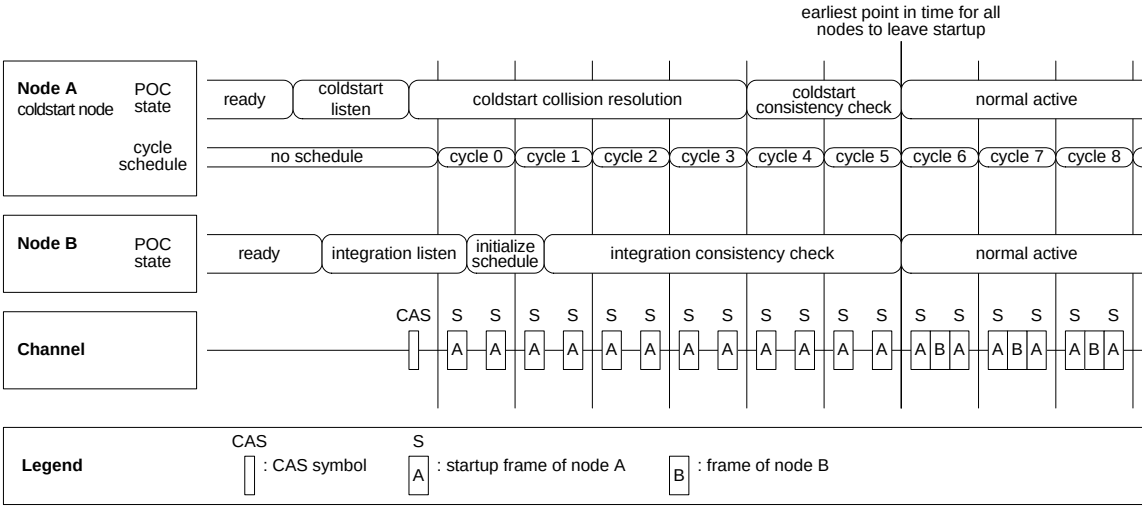

Figure 7-10: Example of state transitions for a fault-free startup in a TT-L cluster.[97]

From cycle zero on, the node transmits its two startup frames.

As no collision with startup frames or CAS symbols transmitted by other nodes can occur (in the fault-free case), the TT-L coldstart node proceeds straight through the *POC:coldstart collision resolution* state. In the following two cycles, the TT-L coldstart node remains in *POC:coldstart consistency check* and always proceeds on to *POC:normal active*, as the two startup frames it provides are sufficient to initialize and maintain the TT-L cluster (see SDL macro COLDSTART_CONSISTENCY_CHECK in Figure 7-17).

7.2.4.4 Path of a TT-E coldstart node

Figure 7-11 depicts a system topology that is used to describe the path followed by TT-E coldstart nodes. In this figure, which depicts two independent time gateways, nodes M, N, P and Q are connected to the time source cluster (a cluster using the TT-D synchronization mode) and nodes A, B, and C are connected to the time sink cluster. In the TT-D cluster, nodes N, P, and Q are coldstart nodes.

Figure 7-12 depicts the path of nodes A and B, the coldstart nodes of the TT-E cluster. As Figure 7-13 shows, the behavior of a node following this path is straightforward.

[97] Please note that several simplifications have been made within this diagram to make it more accessible, e.g. the state transitions do not occur on cycle change, but well before that (see Chapter 8).

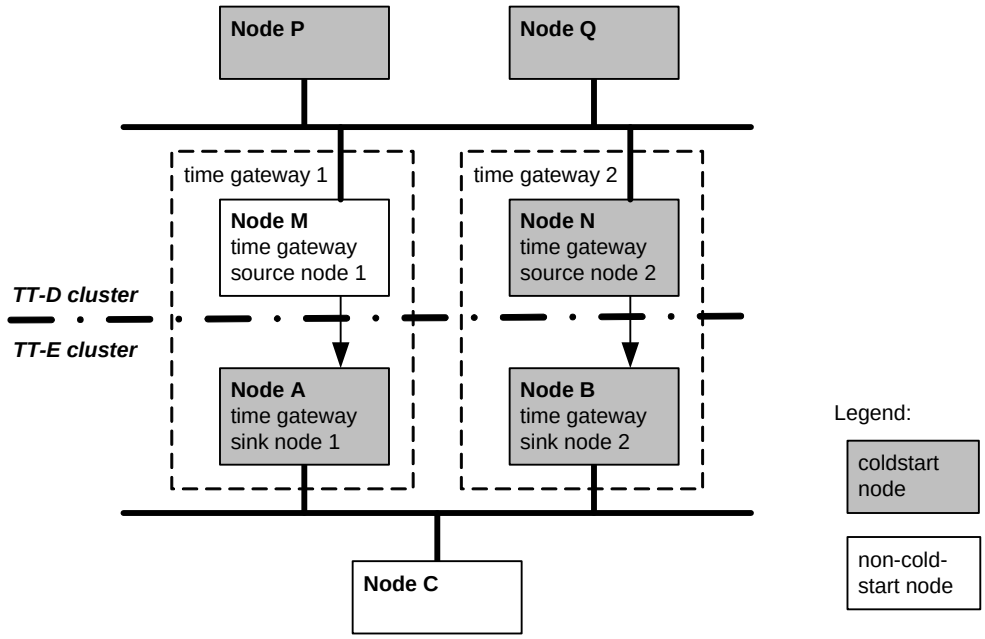

Figure 7-11: Topology for the startup example of a TT-E cluster shown in Figure 7-12.

The node first starts the CSP, MAC and FSP processes and then awaits the first *cycle start* signal; receiving this, the node directly continues into *POC:normal active*. The *cycle start* signal is generated as soon as the (just started) CSP process has managed to synchronize itself onto the time gateway source (see Figure 8-5).

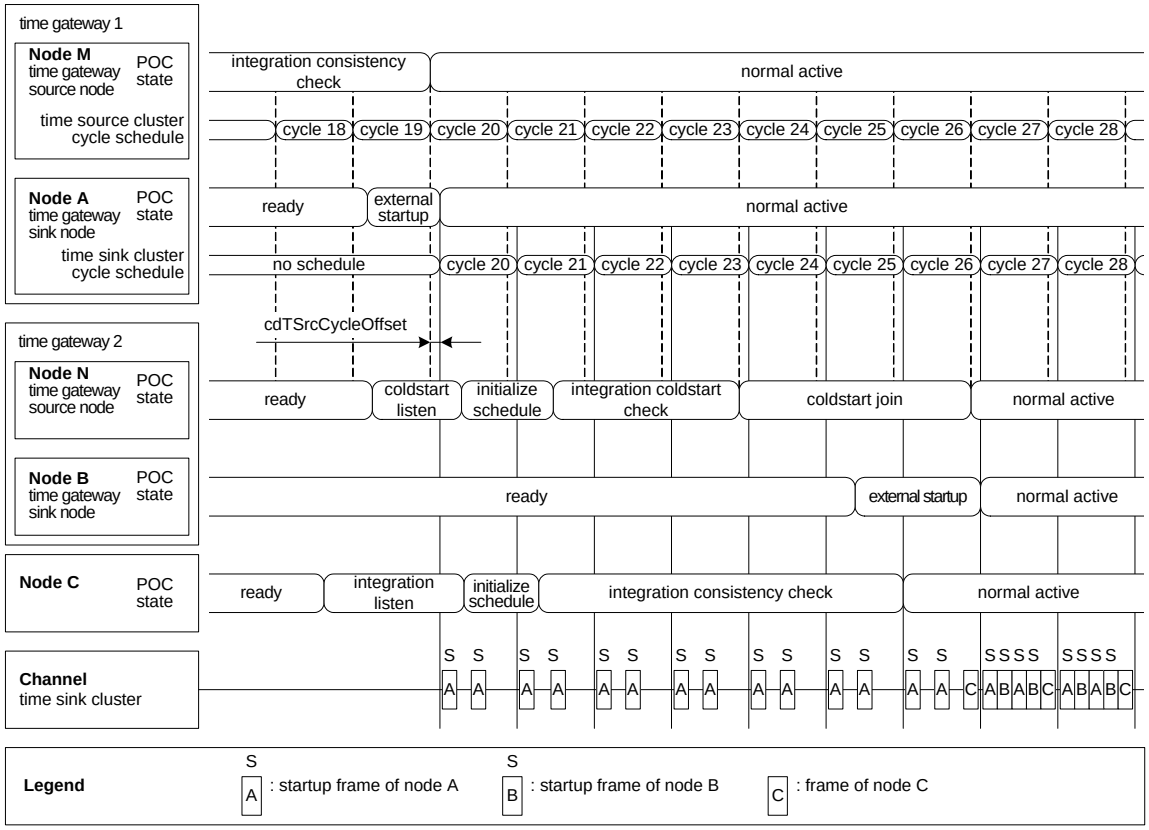

Figure 7-12: Example of state transitions for a fault-free startup in a TT-E cluster.[98]

Node A in Figure 7-12 enters *POC:normal active* very shortly after its time gateway source node (node M) has entered *POC:normal active* itself. The time gateway source node has provided the node A with all relevant information about clock correction values during cycles 18 and 19, but as node M was not yet in *POC:normal active*, node A could not proceed. Node A uses the cycle counter value provided by the time source gateway node and therefore directly starts with the cycle number 20, the same cycle number the time source cluster currently uses. The cycle schedules of Figure 7-12 have been drawn slightly offset to one another to symbolize the fixed offset of *cdTSrcCycleOffset* microticks between the cluster schedules. The second TT-E coldstart node, node B, also enters *POC:normal active* as soon as it has received all relevant terms from its time gateway source node (node N) and does not verify its view on the schedule against the already present startup frames. Node B is not required for node C to complete its startup.

[98] Please note that several simplifications have been made within this diagram to make it more accessible, e.g. the state transitions do not occur on cycle change, but well before that (see Chapter 8).

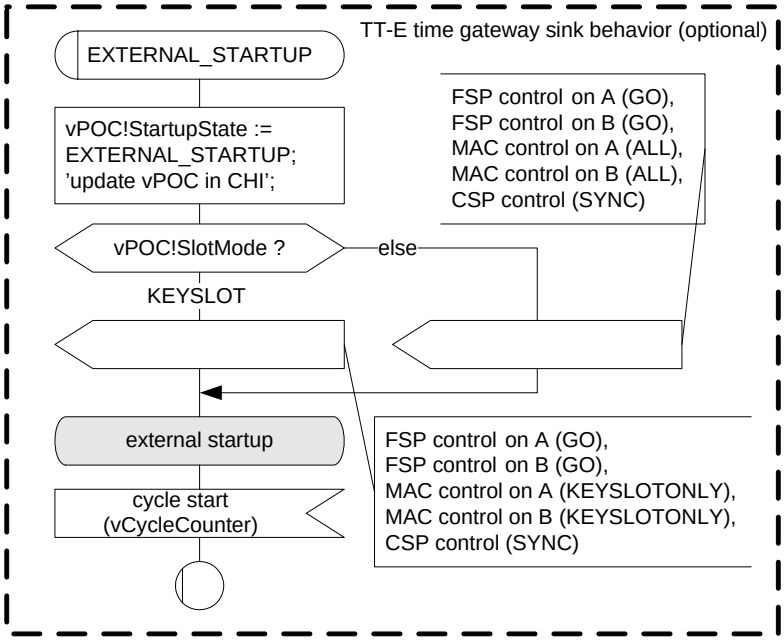

Figure 7-13: External startup state [POC].

7.2.4.5 Path of a non-coldstart node

Node C in Figure 7-9, Node B in Figure 7-10, and Node C in Figure 7-12 follow this path.

When a non-coldstart node enters startup, it listens to its attached channels and tries to receive FlexRay frames (see SDL macro INTEGRATION_LISTEN in Figure 7-22).

If communication[99] is received, it tries to integrate to a transmitting coldstart node. It tries to receive a valid pair of startup frames to derive its schedule and clock correction from the coldstart node (see Chapter 8 and see SDL macro INITIALIZE_SCHEDULE in Figure 7-19).

In the following double cycles, it tries to find at least two startup frames that fit into its own schedule. In a TT-D cluster these frames will come from different coldstart nodes. If this fails, or if clock correction signals an error, the node aborts the integration attempt and tries again.

After receiving two valid startup frame pairs with different frame IDs for two consecutive double cycles, the node leaves startup and enters operation.

For TT-D clusters, this means that the non-coldstart node leaves startup at least two cycles later than the node that initiated the coldstart. Effectively, all nodes of a TT-D cluster can leave startup at the end of cycle 7, just before entering cycle 8 (see Figure 7-9 and SDL macro INTEGRATION_CONSISTENCY_CHECK in Figure 7-23). For TT-D and TT-E clusters, this time is reduced by one double cycle, as the two necessary startup frames are present as much earlier (see Figure 7-10 and Figure 7-12).

[99] See Chapter 8 for exact definition.

7.2.4.6 The *POC:coldstart listen* state

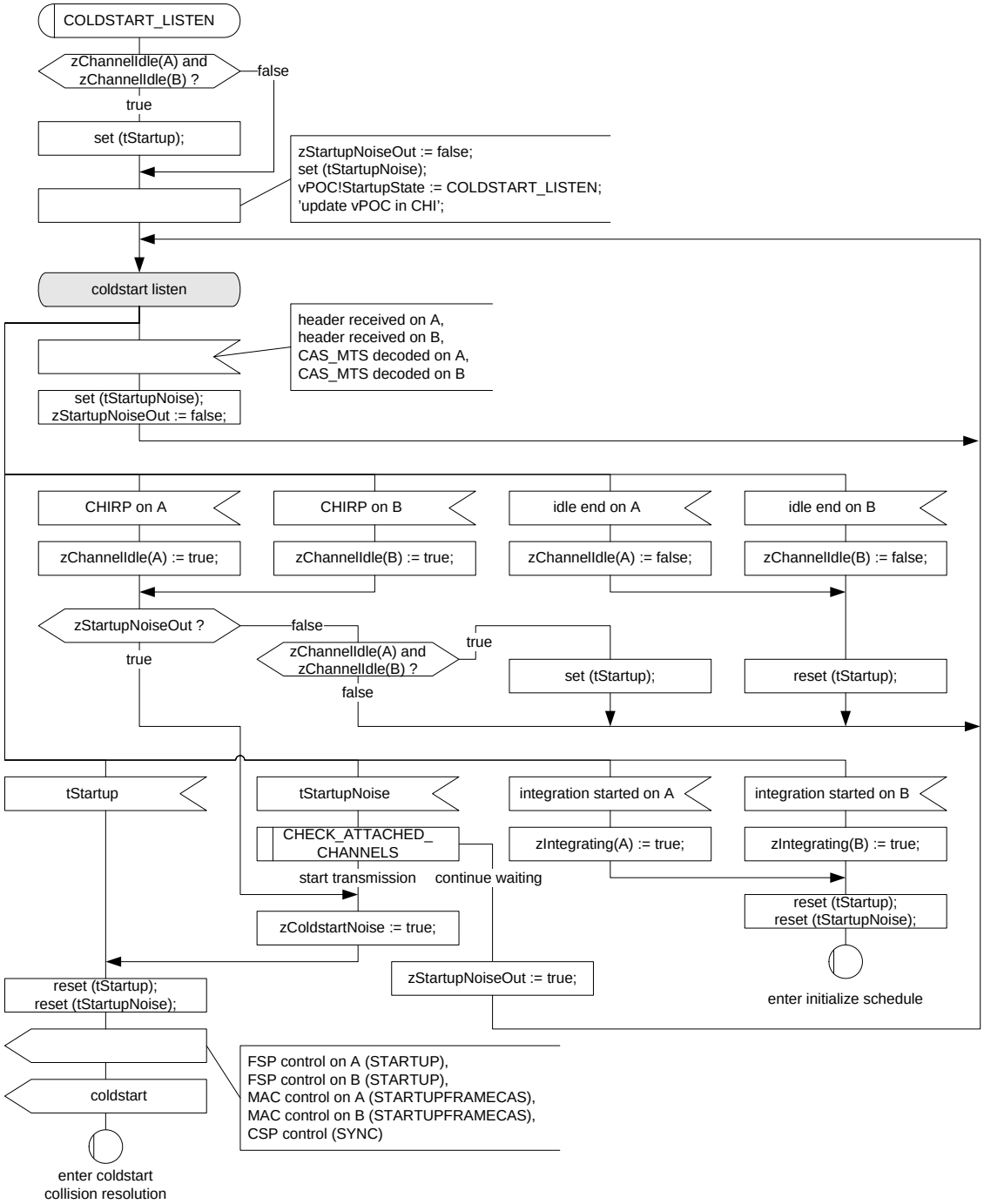

Figure 7-14: Transitions from the *POC:coldstart listen* state [POC].[100]

[100] Note that if all attached channels are stuck continuously active low POC will remain in the startup until the host commands it to a different state.

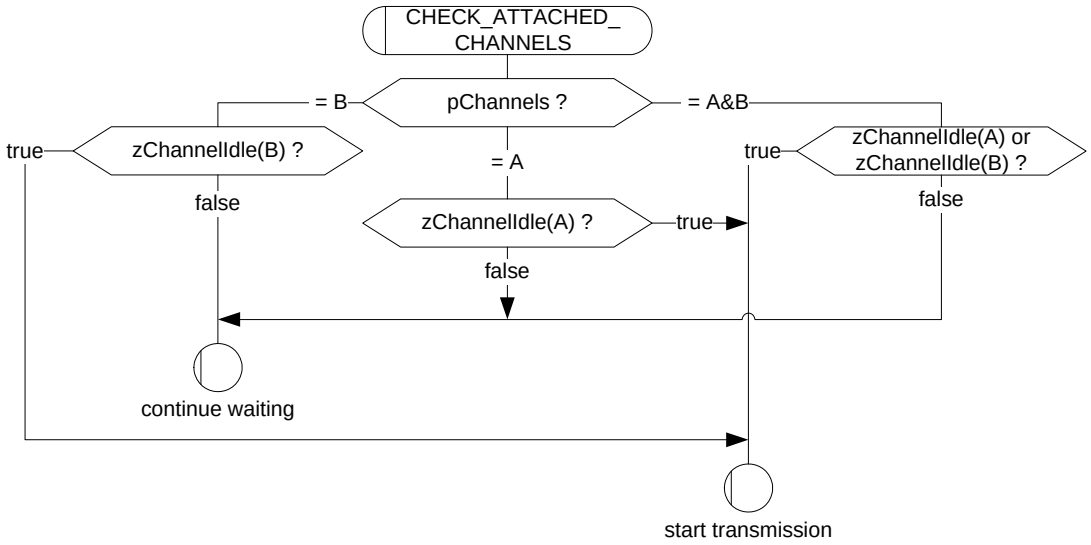

Figure 7-15: Macro CHECK_ATTACHED_CHANNELS [POC].

A coldstart node still allowed[101] to initiate a coldstart enters the *POC:coldstart listen* state before actually performing the coldstart. In this state the coldstart node tries to detect ongoing frame transmissions and coldstart attempts.

This state is left and the *POC:initialize schedule* state is entered as soon as a valid startup frame has been received (see Chapter 8 for details of this mechanism), as the node tries to integrate on the node that has transmitted this frame.

When neither CAS symbols nor frame headers can be detected for a predetermined time duration,the node initiates the coldstart and enters the *POC:coldstart collision resolution* state. The amount of time that has to pass before a coldstart attempt may be performed is defined by the two timers *tStartup* and *tStartupNoise*. The timer *tStartup* expires quickly, but is stopped whenever a channel is active (see Chapter 3 for a description of channel states). It is restarted when all attached channels are in idle state. The timer *tStartupNoise* is only restarted by the reception of correctly decoded headers or CAS symbols to guarantee a cluster startup when noise interference is present or if a single channel is permanently busy.

[101] See macro STARTUP_PREPARE in Figure 7-8. The condition '*vRemainingColdstartAttempts* > 1' arises from the necessity of using up one round through the *POC:coldstart collision resolution* state for the collision resolution and needing the second round for actually integrating the other nodes.

7.2.4.7 The *POC:coldstart collision resolution* state

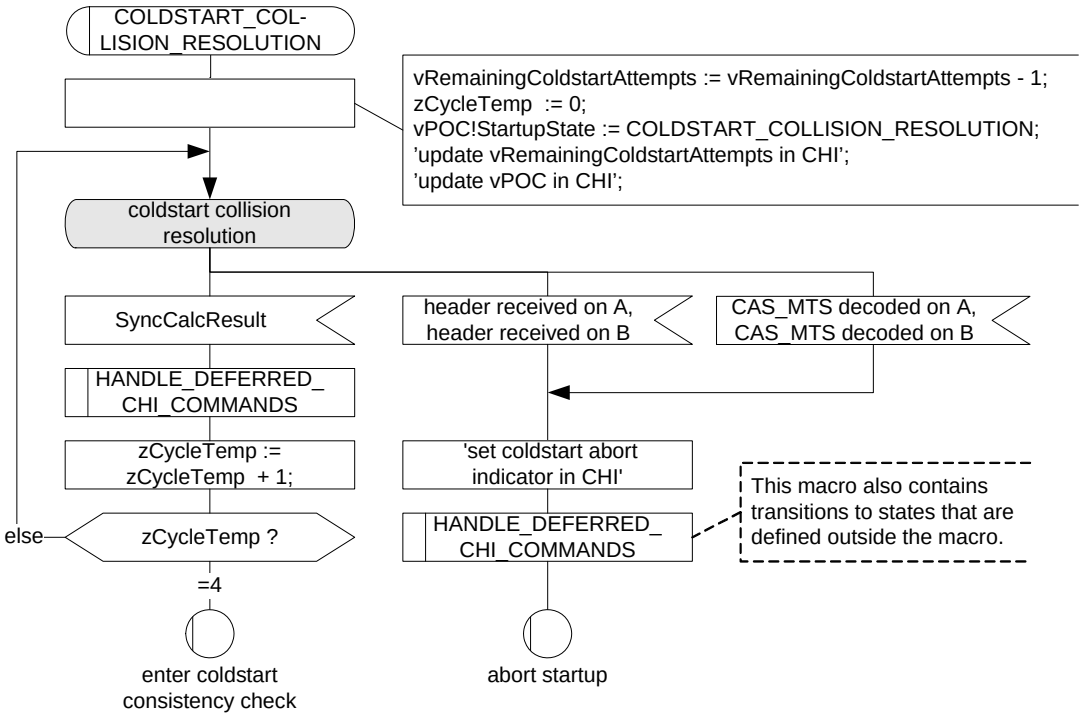

Figure 7-16: Transitions from the *POC:coldstart collision resolution* state [POC].

The purpose of this state is to detect and resolve collisions between multiple simultaneous coldstart attempts of several coldstart nodes. Each entry into this state starts a new coldstart attempt by this node.

The reception of a complete header without coding errors or the reception of a valid CAS symbol causes the communication controller to abort the coldstart attempt. This resolves conflicts between multiple coldstart nodes performing a coldstart attempt at the same time, so only one leading coldstart node remains. In the fault-free case and under certain configuration constraints (see Appendix B) only one coldstart node will proceed to the *POC:coldstart consistency check* state. The other nodes abort startup since they received a frame header from the successful coldstart node.

The number of coldstart attempts that a node is allowed to make is restricted to the initial value of the variable *vRemainingColdstartAttempts*. *vRemainingColdstartAttempts* is reduced by one for each attempted coldstart. A node may enter the *POC:coldstart listen* state only if this variable is larger than one and it may enter the *POC:coldstart collision resolution* state only if this variable is larger than zero. A value of larger than one is required for entering the *POC:coldstart listen* state because one coldstart attempt may be used for performing the collision resolution, in which case the coldstart attempt could fail.

After four cycles in this state, the node enters the *POC:coldstart consistency check* state.

7.2.4.8 The *POC:coldstart consistency check* state

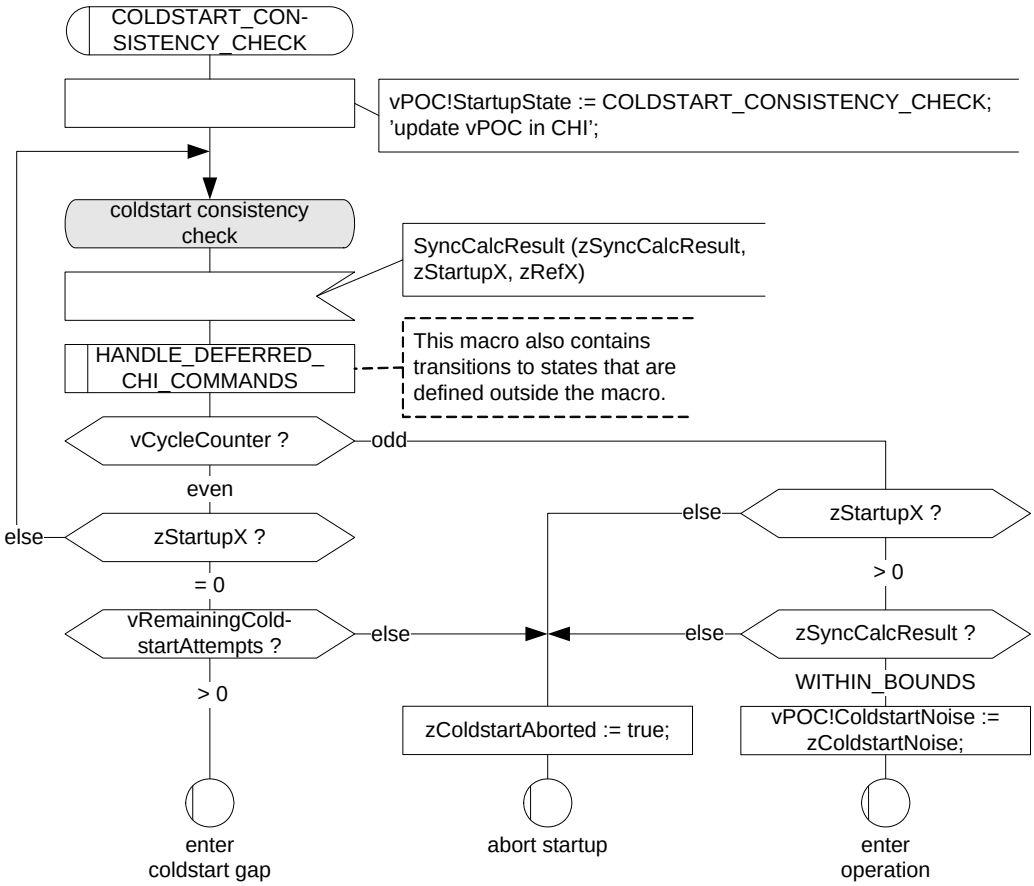

Figure 7-17: Transitions from the *POC:coldstart consistency check* state [POC].[102]

In this state, the leading coldstart node checks whether the frames transmitted by other following coldstart nodes (non-coldstart nodes cannot yet transmit in the fault-free case) fit into its schedule.

If a TT-D coldstart node receives no valid startup frames in the even cycle in this state, it is assumed that the other coldstart nodes were not ready soon enough to initialize their schedule from the first two startup frames sent during the *POC:coldstart collision resolution* state. Therefore, if another coldstart attempt is allowed, the node enters the *POC:coldstart gap* state to wait for the other coldstart nodes to get ready.

If a TT-D coldstart node has received a valid startup frame in the even cycle in this state, but the clock correction signals errors in the odd cycle or no valid pair of startup frames can be received in the double cycle, the node aborts the coldstart attempt.

If a TT-D coldstart node has received a valid pair of startup frames and the clock correction signals no errors the node leaves startup and enters operation (see Chapter 2).

A TT-L coldstart node will enter operation after having remained for two cycles in this state, as it is the sole provider of startup frames of the cluster.

[102] *zStartupX* is *zStartupNodes* in even cycles and *zRxStartupPairs* in odd cycles. *zRefX* is *zRefNode* in even cycles and *zRefPair* in odd cycles. See Figure 8-6 for details.

7.2.4.9 The *POC:coldstart gap* state

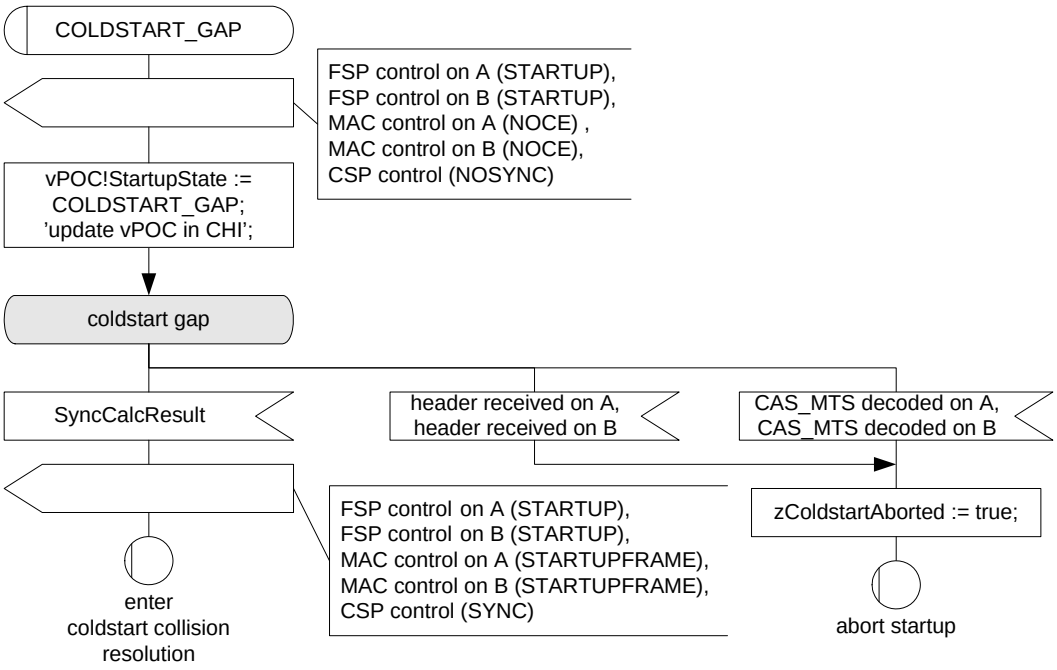

Figure 7-18: Transitions from the *POC:coldstart gap* state [POC].

In the *POC:coldstart gap* state the leading coldstart node stops transmitting its startup frame. This causes all nodes currently integrating on the leading coldstart node to abort their integration attempt.

In the same way as during the *POC:coldstart collision resolution* state, the leading coldstart node aborts the coldstart attempt if it receives a frame header or a valid CAS symbol. If it does not receive either, it proceeds after one cycle by reentering the *POC:coldstart collision resolution* state for another coldstart attempt.

7.2.4.10 The *POC:initialize schedule* state

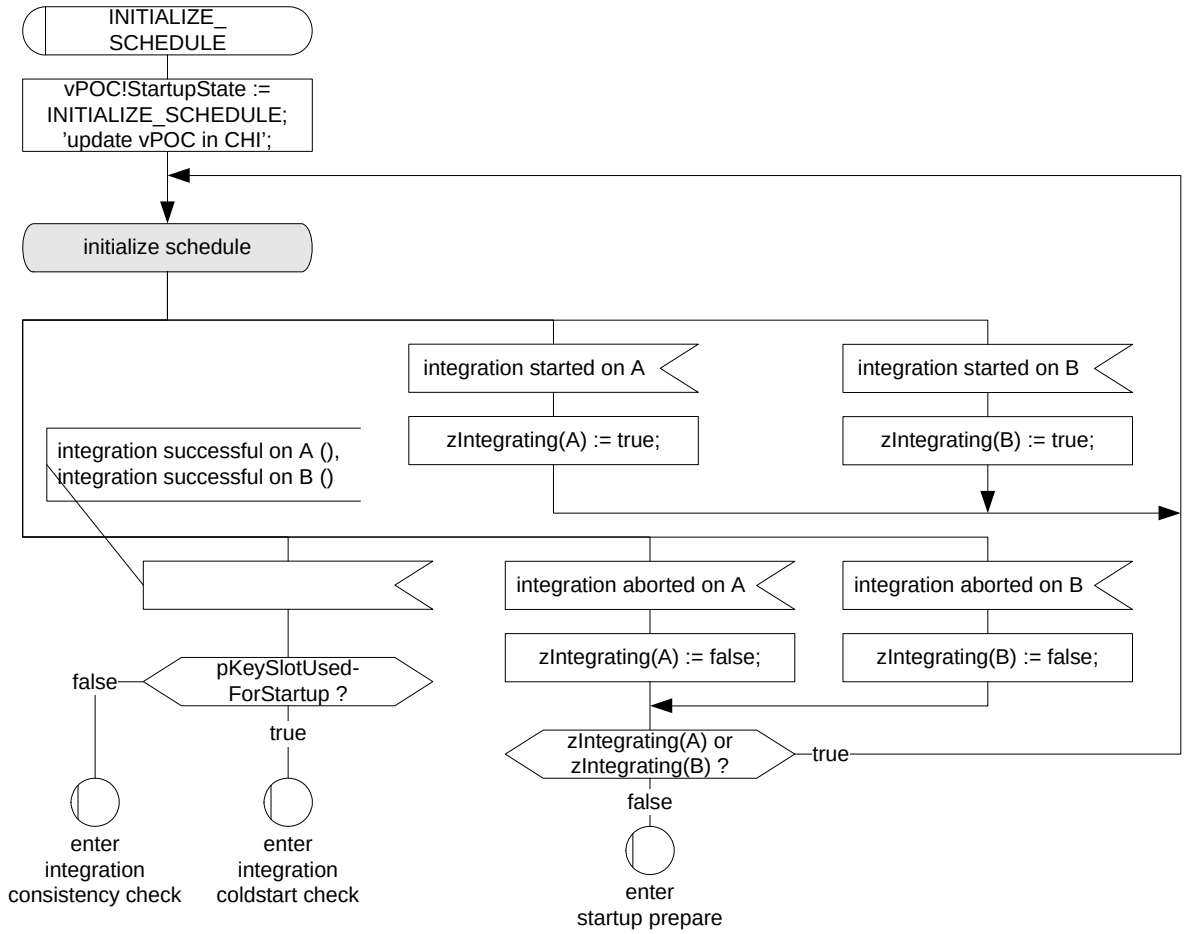

Figure 7-19: Transitions from the *POC:initialize schedule* state [POC].

As soon as a valid startup frame has been received in one of the listen states (see Figure 7-7), the *POC:initialize schedule* state is entered. If clock synchronization successfully receives a matching second valid startup frame and derives a schedule from them (indicated by receiving the signal *integration successful on A* or *integration successful on B*), the POC goes to the *POC:integration coldstart check* state (for coldstart nodes) or the *POC:integration consistency check* state (for non-coldstart nodes).

7.2.4.11 The *POC:integration coldstart check* state

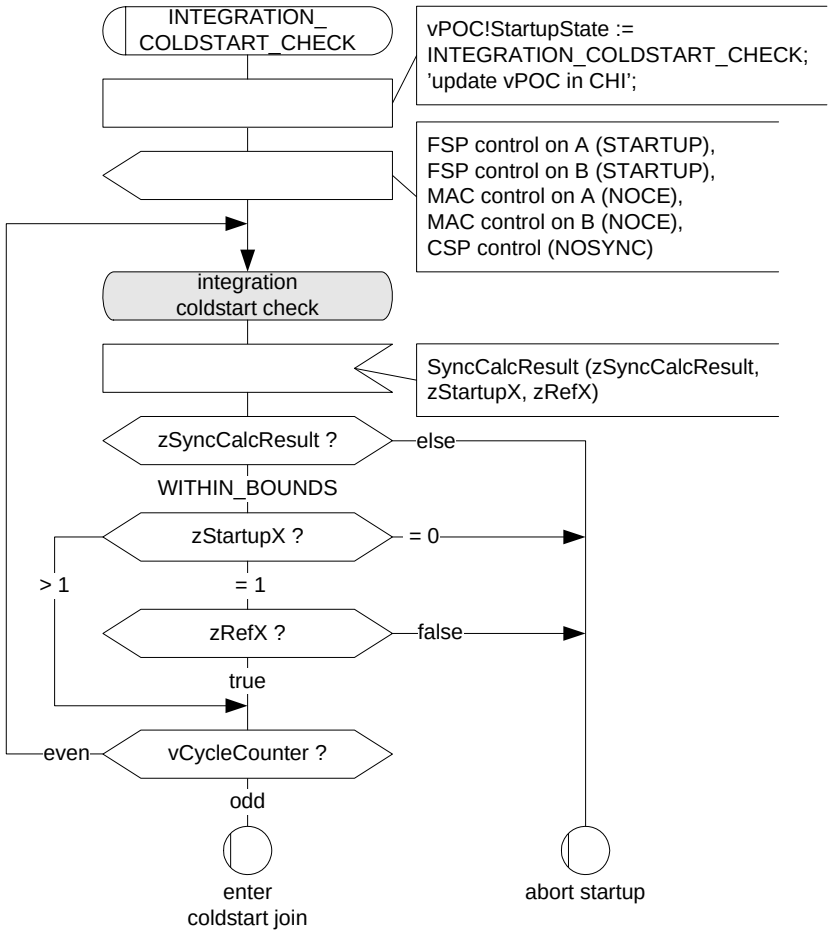

Figure 7-20: Transitions from the *POC:integration coldstart check* state [POC].[103]

Only integrating (following) coldstart nodes pass through this state. In this state it shall be verified that the clock correction can be performed correctly, that at least one coldstart node is still available, and if exactly one coldstart node is available that it is the same coldstart node that was used to initialize the schedule.

The clock correction is activated and if any error is signaled the integration attempt is aborted.

During the first double cycle in this state either two valid startup frame pairs or the startup frame pair of the node that this node has integrated on must be received; otherwise the node aborts the integration attempt.

If at the end of the first double cycle in this state the integration attempt has not been aborted, the *POC:coldstart join* state is entered.

[103] *zStartupX* is *zStartupNodes* in even cycles and *zRxStartupPairs* in odd cycles. *zRefX* is *zRefNode* in even cycles and *zRefPair* in odd cycles. See Figure 8-6 for details.

7.2.4.12 The *POC:coldstart join* state

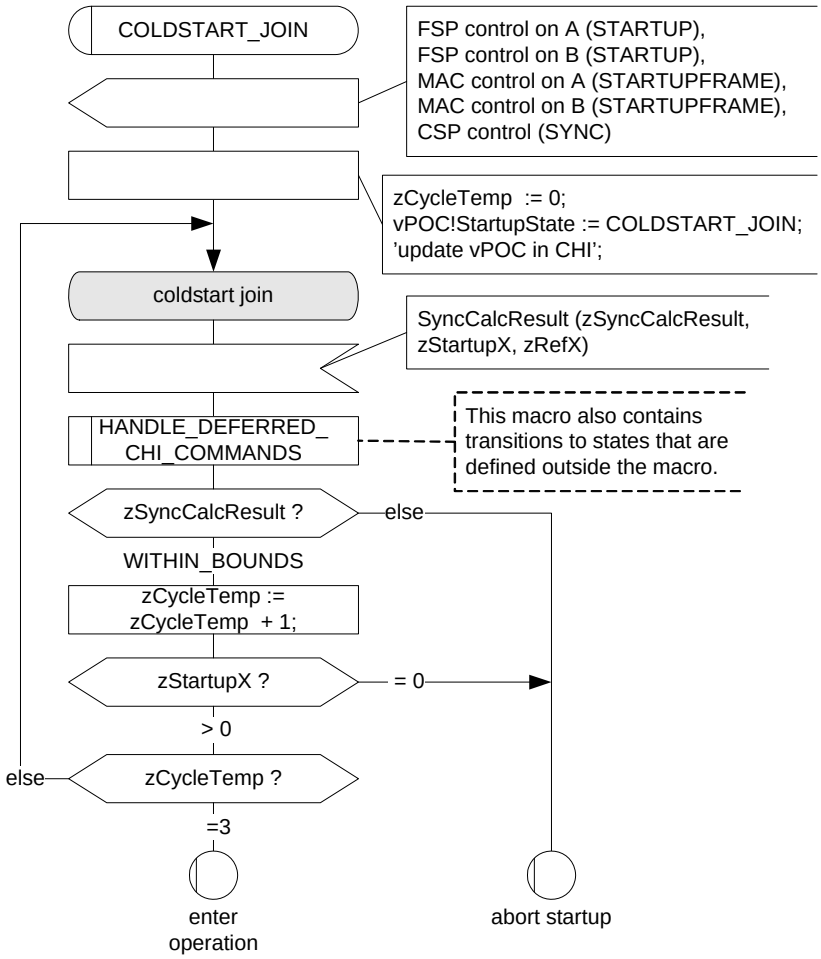

Figure 7-21: Transitions from the *POC:coldstart join* state [POC].[104]

Only following coldstart nodes enter this state. Upon entry they begin transmitting startup frames and continue to do so in subsequent cycles. Thereby, the leading coldstart node and the nodes joining it can check if their schedules agree with each other.

If the clock correction signals any error, the node aborts the integration attempt.

If a node in this state sees at least one valid startup frame during all even cycles in this state and at least one valid startup frame pair during all double cycles in this state, it leaves startup and enters operation (see Chapter 2).

[104] *zStartupX* is *zStartupNodes* in even cycles and *zRxStartupPairs* in odd cycles. *zRefX* is *zRefNode* in even cycles and *zRefPair* in odd cycles. See Figure 8-6 for details.

7.2.4.13 The *POC:integration listen* state

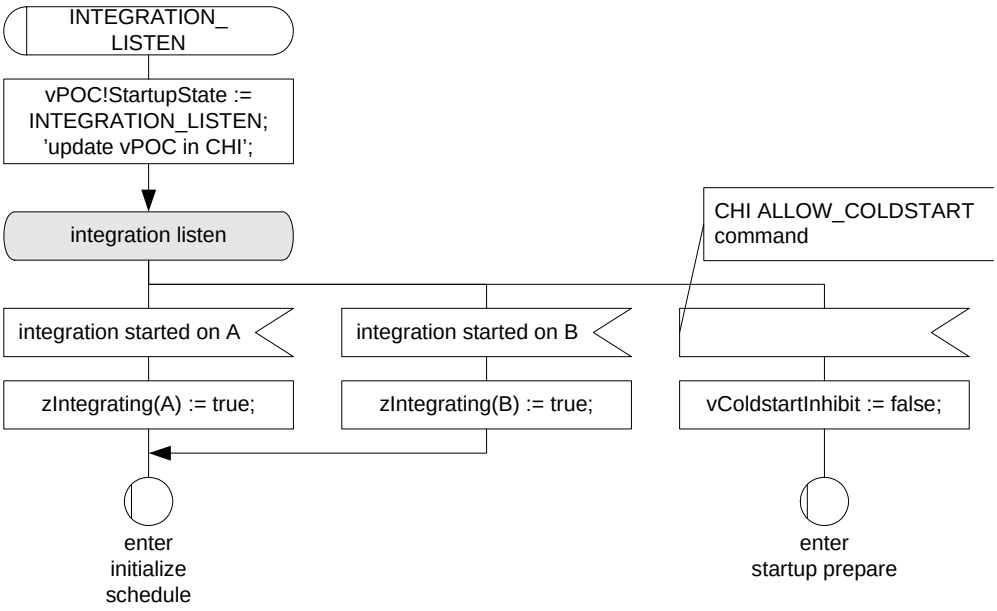

Figure 7-22: Transitions from the *POC:integration listen* state [POC].

In this state the node waits for either a valid startup frame or for the *vColdstartInhibit* variable to be cleared.

If the *vColdstartInhibit* variable is cleared the node reevaluates whether it is allowed to initiate a coldstart and consequently enter the *POC:coldstart listen* state.

7.2.4.14 The *POC:integration consistency check* state

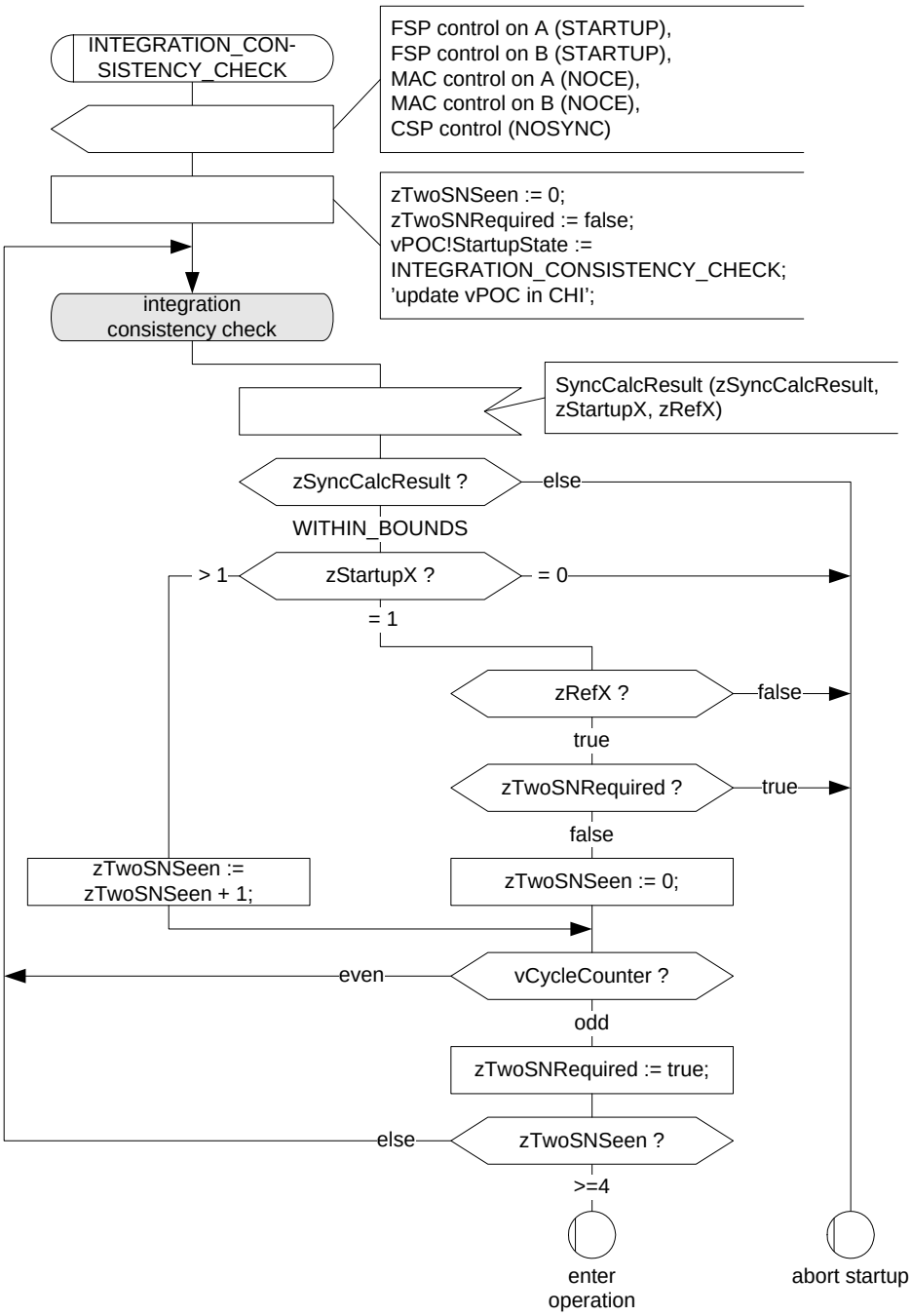

Figure 7-23: Transitions from the *POC:integration consistency check* state [POC].[105]

Only integrating non-coldstart nodes pass through this state. In this state the node verifies that clock correction can be performed correctly and that enough coldstart nodes are sending startup frames that agree with the node's own schedule.

[105] *zStartupX* is *zStartupNodes* in even cycles and *zRxStartupPairs* in odd cycles. *zRefX* is *zRefNode* in even cycles and *zRefPair* in odd cycles. See Figure 8-6 for details.

Clock correction is activated and if any errors are signaled the integration attempt is aborted.

During the first even cycle in this state, either two valid startup frames or the startup frame of the node that this node has integrated on must be received; otherwise the node aborts the integration attempt.

During the first double cycle in this state, either two valid startup frame pairs or the startup frame pair of the node that this node has integrated on must be received; otherwise the node aborts the integration attempt.

After the first double cycle, if less than two valid startup frames are received within an even cycle, or less than two valid startup frame pairs are received within a double cycle, the startup attempt is aborted.

Nodes in this state need to see two valid startup frame pairs for two consecutive double cycles each to be allowed to leave startup and enter operation (see Chapter 2). Consequently, they leave startup at least one double cycle after the node that initiated the coldstart and only at the end of a cycle with an odd cycle number.

Chapter 8
Clock Synchronization

8.1 Introduction

In a distributed communication system every node has its own clock. Because of temperature fluctuations, voltage fluctuations, and production tolerances of the timing source (an oscillator, for example), the internal time bases of the various nodes diverge after a short time, even if all the internal time bases are initially synchronized.

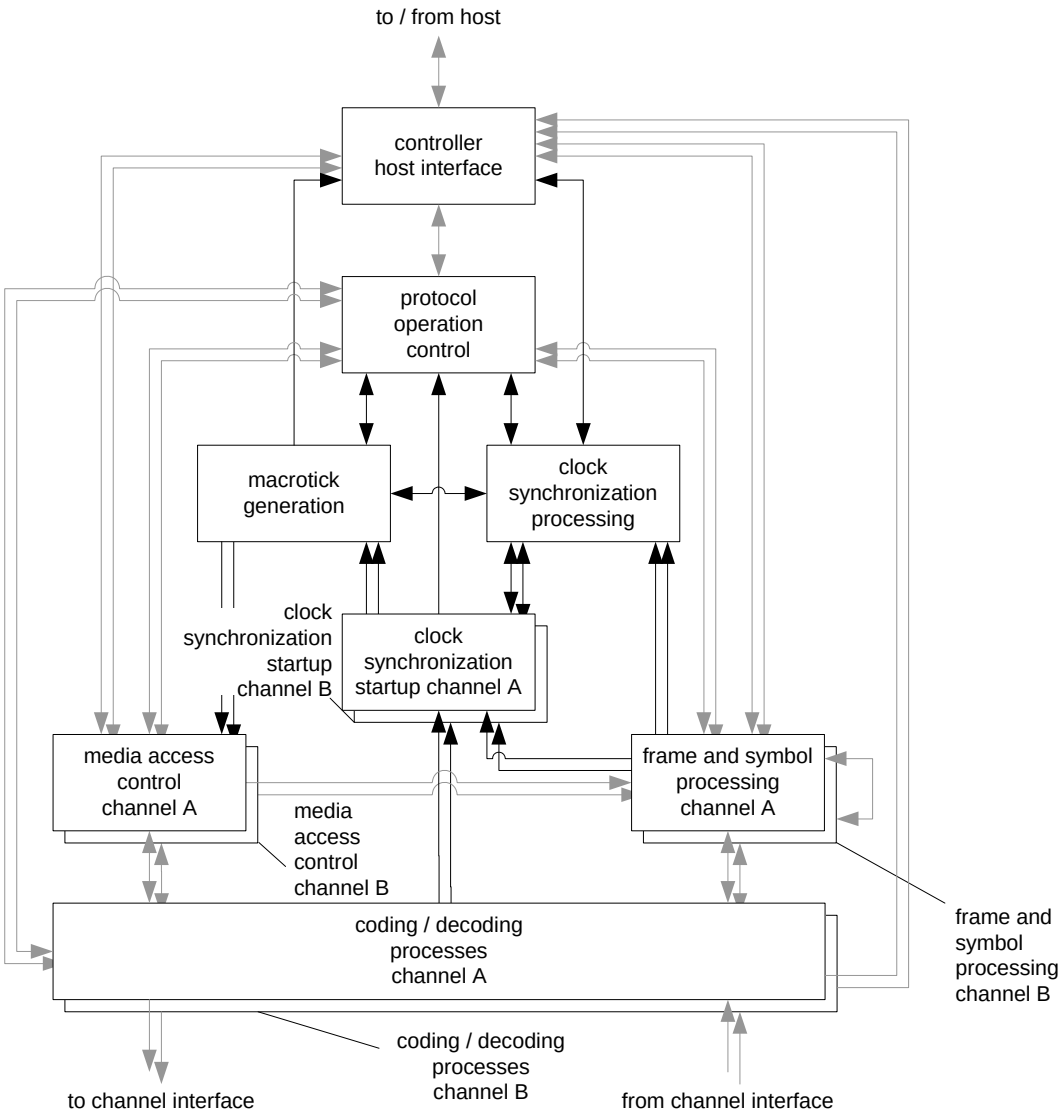

Figure 8-1: Clock synchronization context.

A basic assumption for a time-triggered system is that every node in the cluster has approximately the same view of time and this common global view of time is used as the basis for the communication timing for each node. In this context, "approximately the same" means that the difference between any two nodes' views of the global time is within a specified tolerance limit. The maximum value of this difference is known as the precision.

The FlexRay protocol uses a distributed clock synchronization mechanism in which each node individually synchronizes itself to the cluster by observing the timing of transmitted sync frames from other nodes. A fault-tolerant algorithm is used.

The relationship between the clock synchronization processes and the other protocol processes is depicted in Figure 8-1[106].

8.2 Time representation

8.2.1 Timing hierarchy

The time representation inside a FlexRay node is based on cycles, macroticks and microticks. A macrotick is composed of an integer number of microticks. A cycle is composed of an integer number of macroticks (see Figure 8-2).

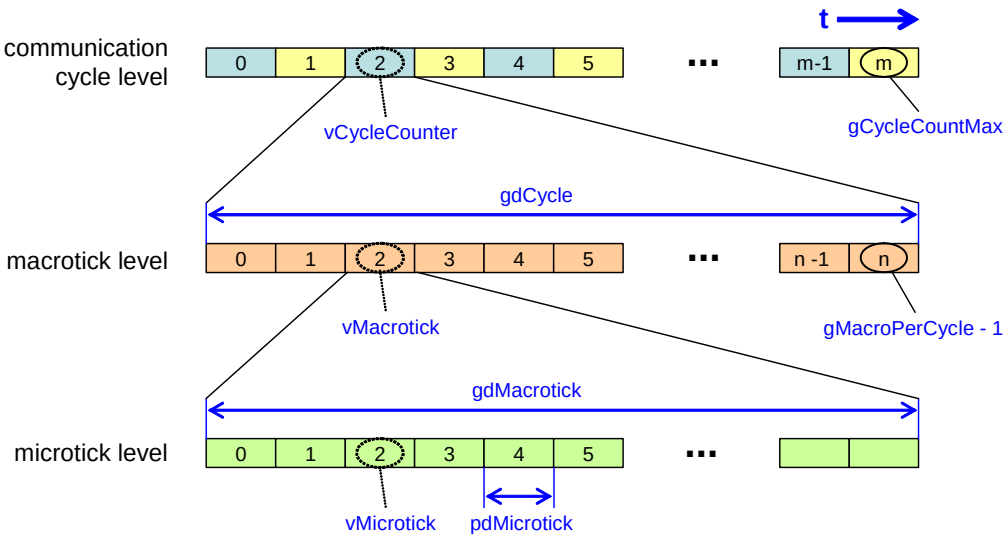

Figure 8-2: Timing hierarchy.

Microticks are time units derived directly from the communication controller's (external) oscillator clock tick, optionally making use of a prescaler. Microticks are controller-specific units. They may have different durations in different controllers. The granularity of a node's internal local time is a microtick.

The macroticks are synchronized on a cluster-wide basis. Within tolerances, the duration of a macrotick is identical throughout all synchronized nodes in a cluster. The duration of each local macrotick is an integer number of microticks; the number of microticks per macrotick may, however, differ from macrotick to macrotick within the same node. The number of microticks per macrotick may also differ between nodes, and depends on the oscillator frequency and the prescaler. Although any given macrotick consists of an integral number of microticks, the average duration of all macroticks in a given cycle may be non-integral (i.e., it may consist of a whole number of microticks plus a fraction of a microtick).[107]

[106] The dark lines represent data flows between mechanisms that are relevant to this chapter. The lighter gray lines are relevant to the protocol, but not to this chapter.

A cycle consists of an integer number of macroticks. The number of macroticks per cycle shall be identical in all nodes in a cluster, and remains the same from cycle to cycle. At any given time all nodes should have the same cycle number (except at cycle boundaries as a result of imperfect synchronization in the cluster).[108]

8.2.2 Global and local time

The global time of a cluster is the general common understanding of time inside the cluster. The FlexRay protocol does not have an absolute or reference global time; every node has its own local view of the global time.

The local time is the time of the node's clock and is represented by the variables *vCycleCounter*, *vMacrotick*, and *vMicrotick*. *vCycleCounter* and *vMacrotick* shall be visible to the application. The update of *vCycleCounter* at the beginning of a cycle shall be atomic with the update of *vMacrotick*.[109]

The local time is based on the local view of the global time. Every node uses the clock synchronization algorithm to attempt to adapt its local view of time to the global time.

The precision of a cluster is the maximum difference between the local times of any two synchronized nodes in the cluster.

8.2.3 Parameters and variables

vCycleCounter is the (controller-local) cycle number and is incremented by one at the beginning of each communication cycle. *vCycleCounter* ranges from 0 to *gCycleCountMax*. When *gCycleCountMax* is reached, the cycle counter *vCycleCounter* shall be reset to zero in the next communication cycle instead of being incremented.

vMacrotick represents the current value of the (controller-local) macrotick and ranges from 0 to (*gMacroPerCycle* - 1). *gMacroPerCycle* defines the (integer) number of macroticks per cycle.

vMicrotick represents the current value of the (controller-local) microtick.

```
syntype
      T_Macrotick = Integer
endsyntype;
syntype
      T_Microtick = Integer
endsyntype;
```

Definition 8-1: Formal definition of T_Macrotick and T_Microtick.

The FlexRay "timing" will be configured by:

- *gCycleCountMax*,
- *gMacroPerCycle*, and
- two of the three parameters *pMicroPerCycle*, *gdCycle* and *pdMicrotick*. *pMicroPerCycle* is the node specific number of microticks per cycle, *gdCycle* is the cluster wide duration of one communication cycle, and *pdMicrotick* is the node specific duration of one microtick. The relation between these three parameters is described in section B.4.15.

[107] This is true even for the nominal (uncorrected) average duration of a macrotick (for example 6000 microticks distributed over 137 macroticks).
[108] The cycle number discrepancy is at most one, and lasts no longer than the precision of the system.
[109] An atomic action is an action where no interruption is possible.

8.3 Synchronization process

Clock synchronization consists of two main concurrent processes. The macrotick generation process (MTG) controls the cycle and macrotick counters and applies the rate and offset correction values. This process is explained in detail in section 8.7. The clock synchronization process (CSP) performs the initialization at cycle start, the measurement and storage of deviation values, and the calculation of the offset and the rate correction values. Figure 8-3 illustrates the timing relationship between these two processes and the relationship to the media access schedule.

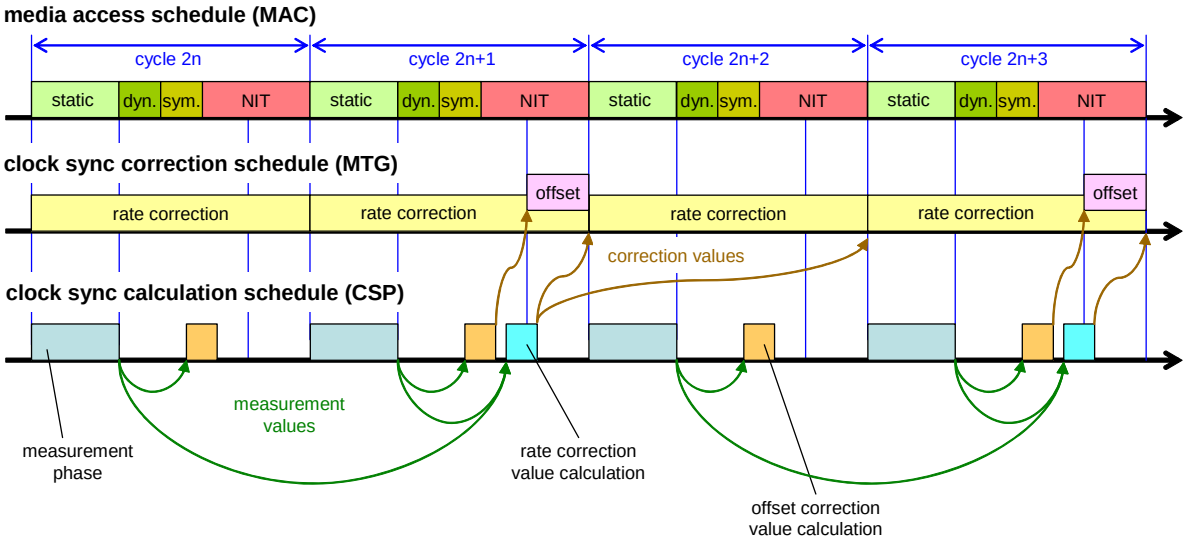

Figure 8-3: Timing relationship between clock synchronization, media access schedule, and the execution of clock synchronization functions.

The primary task of the clock synchronization function is to ensure that the time differences between the nodes of a cluster stay within the precision. Two types of time differences between nodes can be distinguished:

- Offset (phase) differences and
- Rate (frequency) differences.

Methods are known to synchronize the local time base of different nodes using offset correction or rate correction. FlexRay uses a combination of both methods. The following conditions must be fulfilled:

- Rate correction and offset correction shall be done in the same way in all nodes. Rate correction shall be performed over the entire cycle.
- Offset correction shall be performed only during the NIT in the odd communication cycle, starts at *pOffsetCorrectionStart*, and must be finished before the start of the next communication cycle.
- Offset changes (implemented by synchronizing the start time of the cycle) are described by the variable *zOffsetCorrection*. *zOffsetCorrection* indicates the number of microticks that are added to the offset correction segment of the network idle time. *zOffsetCorrection* may be negative. The value of *zOffsetCorrection* is determined by the clock synchronization algorithm. The calculation of *zOffsetCorrection* takes place every cycle but a correction is only applied at the end of odd communication cycles. The calculation of *zOffsetCorrection* is based on values measured in a single

communication cycle. Although the SDL indicates that this computation cannot begin before the NIT, an implementation may start the computation of this parameter within the dynamic segment or symbol window as long as the reaction to the computation (update of the CHI and transmission of the *SyncCalcResult* and *offset calc ready* signals) is delayed until the NIT. The calculation must be complete before the offset correction phase begins.

- Rate (frequency) changes are described by the variable *zRateCorrection*. *zRateCorrection* is an integer number of microticks that are added to the configured number of microticks in a communication cycle (*pMicroPerCycle*)[110]. *zRateCorrection* may be negative. The value of *zRateCorrection* is determined by the clock synchronization algorithm and is only computed once per double cycle. The calculation of *zRateCorrection* takes place following the static segment in an odd cycle. The calculation of *zRateCorrection* is based on the values measured in an even-odd double cycle. Although the SDL indicates that this computation cannot begin before the NIT, an implementation may start the computation of this parameter within the dynamic segment or symbol window as long as the reaction to the computation (update of the CHI and transmission of the *SyncCalcResult* and *rate calc ready* signals) is delayed until the NIT. The calculation must be completed before the next even cycle begins.

The following data types will be used in the definition of the clock synchronization process:

```
newtype T_EvenOdd
      literals even, odd;
endnewtype;
syntype
      T_Deviation = T_Microtick
endsyntype;
```

Definition 8-2: Formal definition of T_EvenOdd and T_Deviation.

The protocol operation control (POC) process sets the operating mode for the clock synchronization process (CSP) (Figure 8-5) into one of the following modes:

1. In the STANDBY mode the clock synchronization process is effectively halted.
2. In the NOSYNC mode CSP performs clock synchronization under the assumption that it is not transmitting sync frames (i.e., it does not include its own clock in the clock correction computations).
3. In the SYNC mode CSP performs clock synchronization under the assumption that it is transmitting sync frames (i.e., it includes its own clock in the clock correction computations).

Definition 8-3 gives the formal definition of the CSP operating modes.

```
newtype T_CspMode
      literals STANDBY, NOSYNC, SYNC;
endnewtype;
newtype T_SyncCalcResult
      literals WITHIN_BOUNDS, EXCEEDS_BOUNDS, MISSING_TERM;
endnewtype;
```

Definition 8-3: Formal definition of T_CspMode and T_SyncCalcResult.

After the POC sets the CSP mode to something other than STANDBY, the CSP waits in the *CSP:wait for startup* state until the POC forces the node to a coldstart or to integrate into a cluster. The startup procedure, including its initialization and interaction with other processes, is described in the macro INTEGRATION_CONTROL, which is explained in section 8.4.

Before further explanation of the processes an array is defined (Definition 8-4) which is used to store the frame IDs of the sync frames that are considered in the clock correction process.

[110] *pMicroPerCycle* is the configured number of microticks per communication cycle without correction.

```
syntype T_ArrayIndex = Integer
      constants 1 : cSyncFrameIDCountMax
endsyntype;

syntype T_SyncFrameIDCount = Integer
      constants 0 : cSyncFrameIDCountMax
endsyntype;

newtype T_FrameIDTable
      Array(T_ArrayIndex, T_FrameID)
endnewtype;
```

Definition 8-4: Formal definition of T_ArrayIndex, T_SyncFrameIDCount, and T_FrameIDTable.

```
newtype T_SyncExtern
      literals UNSYNC, ACTIVE, PASSIVE;
endnewtype;
```

Definition 8-5: Formal definition of T_SyncExtern.

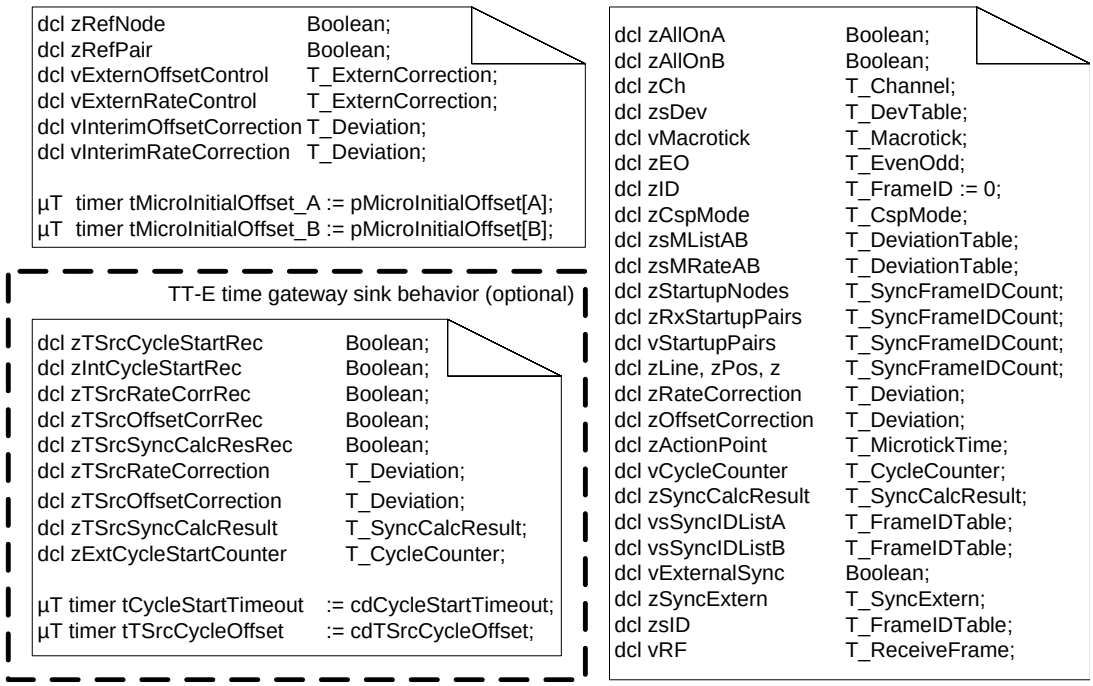

Figure 8-4: Declarations for the clock synchronization process [CSP].

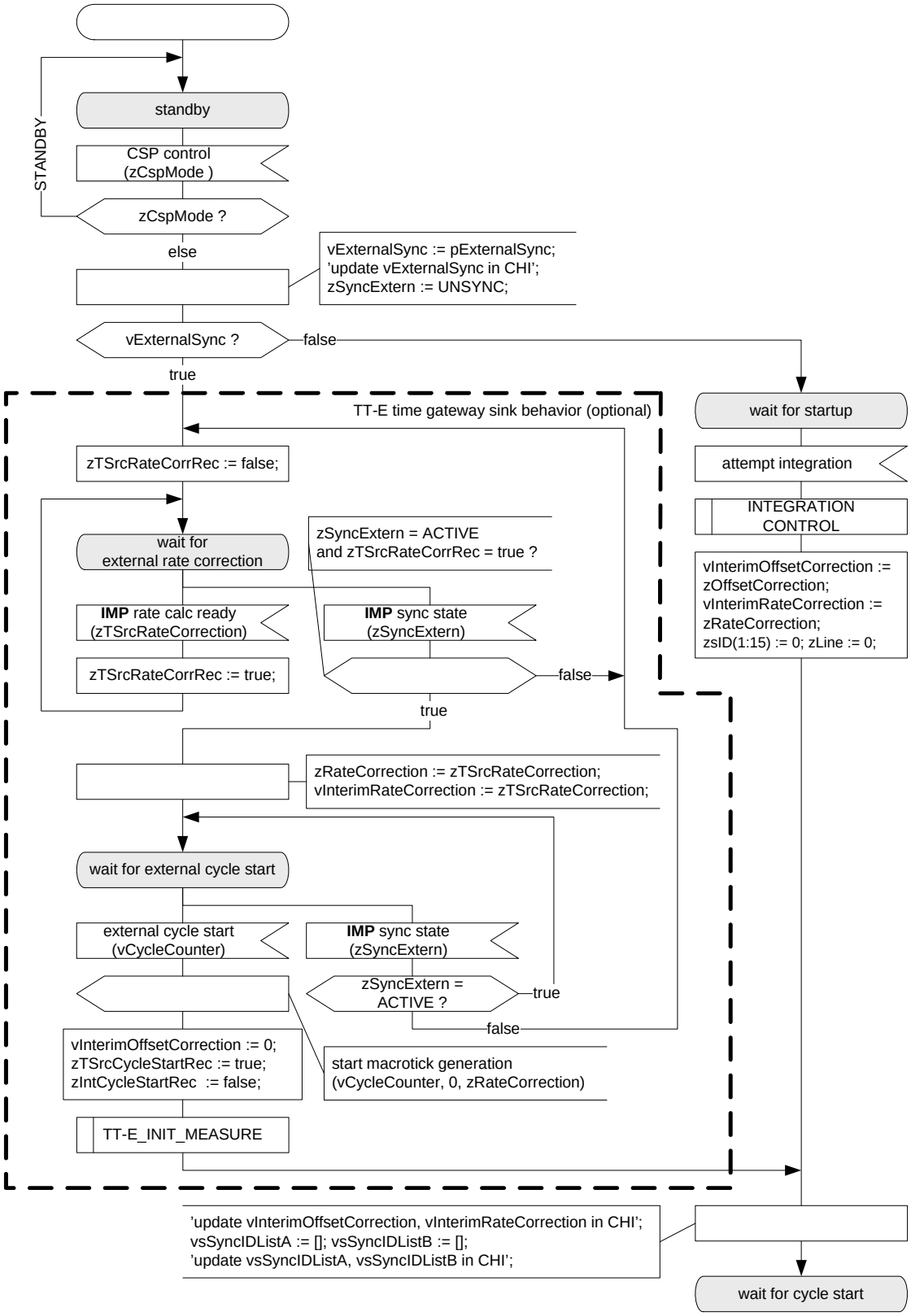

Figure 8-5: Start of the clock synchronization process [CSP].

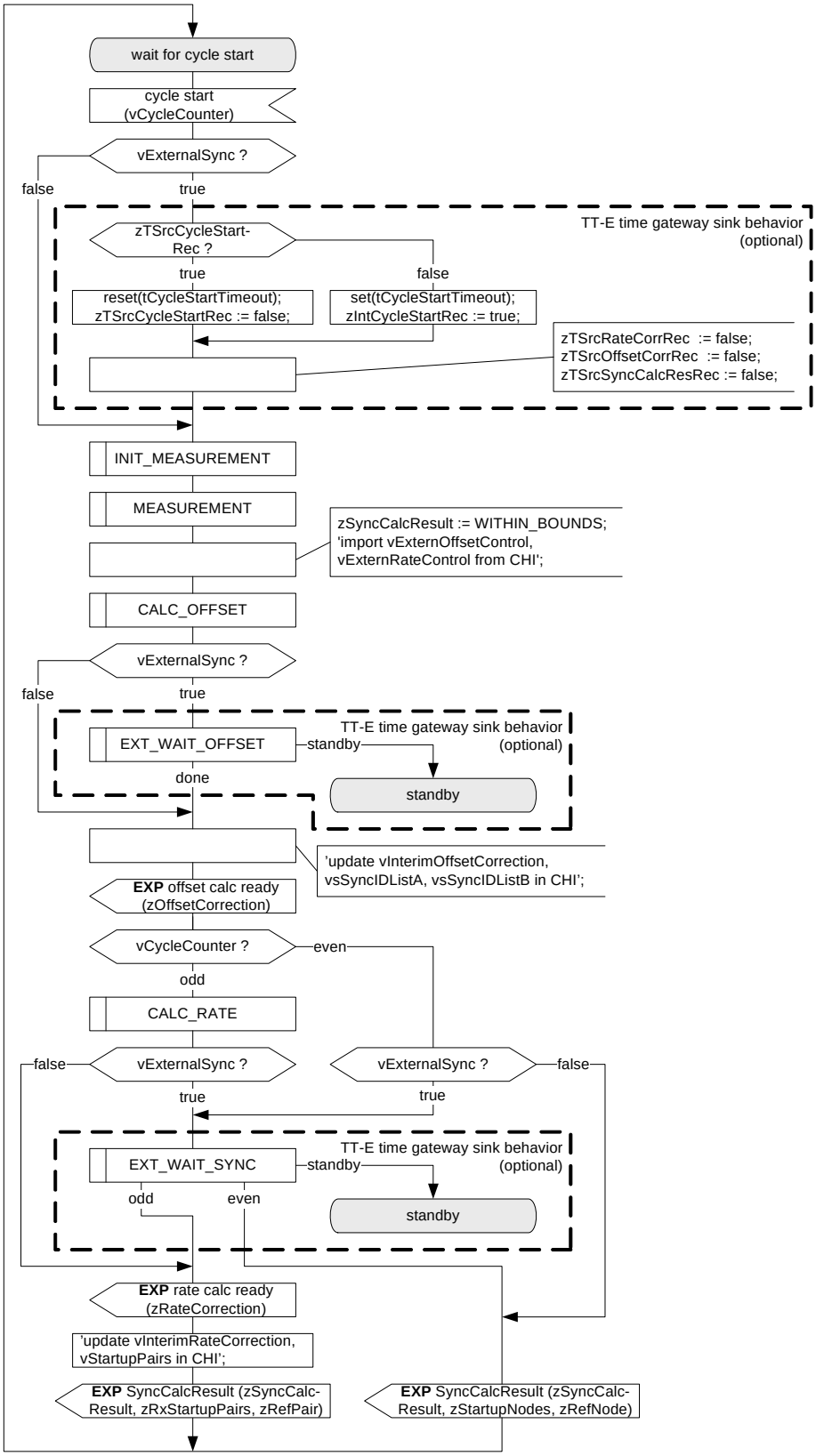

Figure 8-6: Wait for cycle start [CSP].

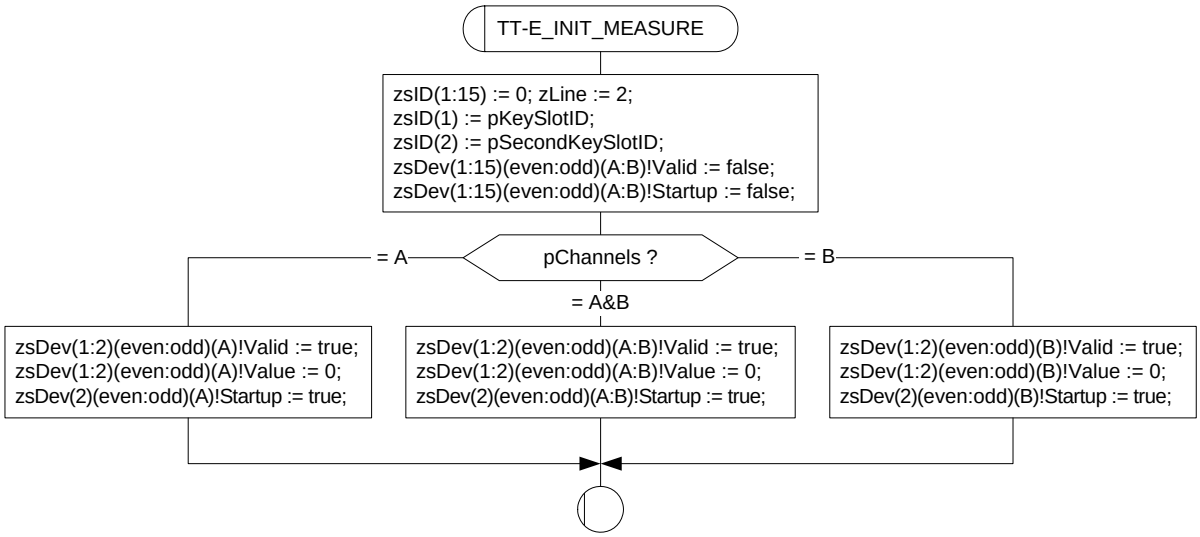

Figure 8-7: Measurement initialization for TT-E coldstart [CSP].

After finishing the startup procedure a repetitive sequence consisting of cycle initialization (Figure 8-13), a measurement phase (Figure 8-14), and offset (Figure 8-17) and rate (Figure 8-18) calculation is executed. All elements of this sequence are described below. The offset calculation will be done every cycle, the rate calculation only in the odd cycles.

The clock synchronization control (Figure 8-8) handles mode changes done by the POC. It also handles process termination requests sent by the POC.

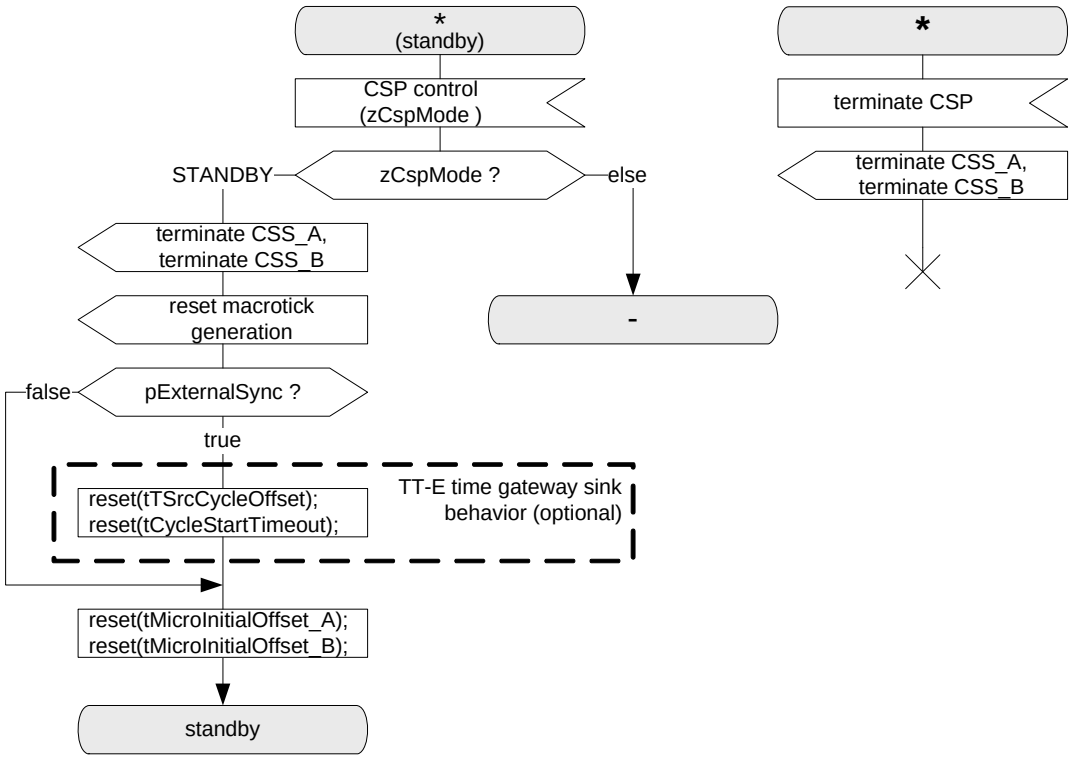

Figure 8-8: Clock synchronization control and termination [CSP].

8.4 Startup of the clock synchronization

The startup of the node's clock synchronization requires

- the initialization and start of the MTG process and
- the initialization and start of the CSP process. This process contains the repetitive tasks of measurement and storage of deviation values and the calculation of the offset and the rate correction values.

There are two ways to start the clock synchronization of a node:

- The node is the leading coldstart node.
- The node adopts the initialization values (cycle counter, clock rate, and cycle start time) of a running coldstart node.

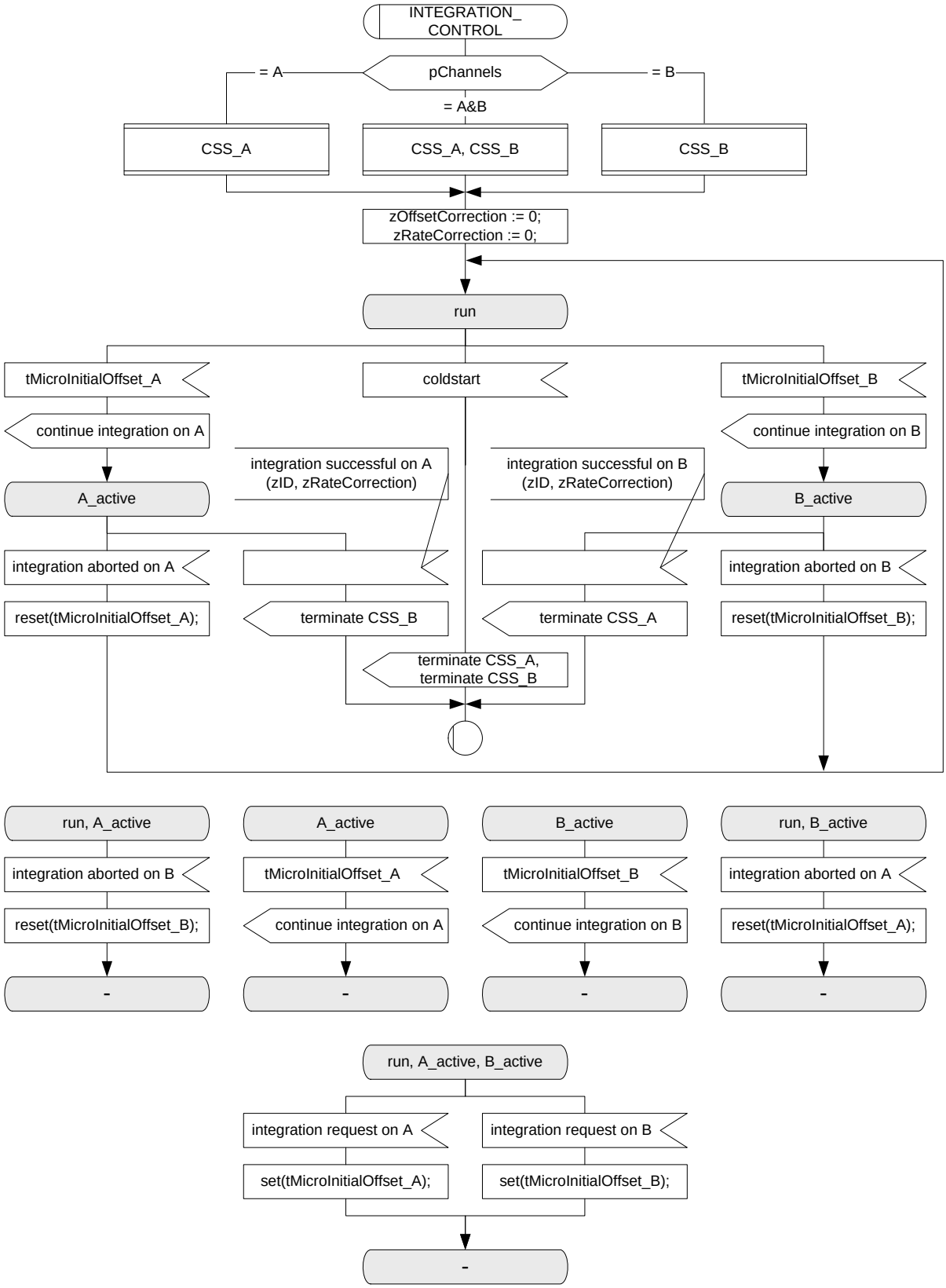

Figure 8-9: Integration control [CSP].

The startup procedure will be entered when the CSP receives the signal *attempt integration* from the POC (see Figure 8-5). The control of the node's startup is described in the INTEGRATION_CONTROL macro depicted in Figure 8-9.

8.4.1 Coldstart startup

If ongoing communication on the channels is not detected the POC may force the node to perform the role of the leading coldstart node of the cluster. This causes the following actions:

- The clock synchronization startup processes on channel A and B (CSS_A, CSS_B) will be terminated.
- The INTEGRATION_CONTROL macro will be left.
- The macrotick generation process (MTG) (Figure 8-25) leaves the *MTG:wait for start* state. Depending on the initialization values, *macrotick* and *cycle start* signals are generated and distributed to other processes.
- The CSP waits for the cycle start.

The CSP and MTG processes continue their schedules until the POC changes the CSP mode to STANDBY or an error is detected.

8.4.2 Integration startup

If ongoing communication is detected during startup, or if the node is not allowed to perform a coldstart, the node attempts to integrate into the timing of the cluster by adopting the rate, the cycle number, and cycle start instant of a coldstart node. To accomplish this, the CSP process (Figure 8-9) instantiates the clock synchronization startup processes for channel A and B (CSS_A, CSS_B).

After their instantiation, the CSS_A process (Figure 8-11) and the CSS_B process wait for a signal from the coding/decoding unit that a potential frame start was detected. The CSS process then takes a timestamp and waits for a signal indicating that a valid even startup frame was received. If no valid even startup frame was received the time stamp will be overwritten with the time stamp of the next potential frame start that is received.

When a valid even startup frame is received the node is able to pre-calculate the initial values for the cycle counter and the macrotick counter. The node then waits for the corresponding odd startup frame. This frame is expected in a time window. When a potential frame start is detected in this time window the *tMicroInitialOffset* timer is started in the INTEGRATION_CONTROL macro. When this timer expires the MTG process (Figure 8-25) is started using the pre-calculated initial values. A second potential frame start inside the time window leads to a restart of the *tMicroInitialOffset* timer. Only one channel can start the MTG process (the initial channel).[111] Between the expiration of the timer *tMicroInitialOffset* and the reception of the complete startup frame, the other channel (the non-initial channel) can not start, stop, or change the MTG process, but it can receive potential frame start events and can start its own *tMicroInitialOffset* timer. The behavior of the CSS process of the non-initial channel is the same as the behavior of the CSS process of the initial channel except that the non-initial channel is unable to start the MTG process and is unable to terminate itself and the CSS process of the other channel.

[111] There is no configuration that selects the channel that starts the MTG process. The process is started by the first channel that receives a potential frame start in the expected time window after reception of a valid even startup frame on the same channel (refer to Figure 8-11).

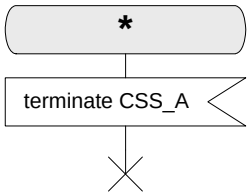

Figure 8-10: Termination of the CSS process [CSS_A].

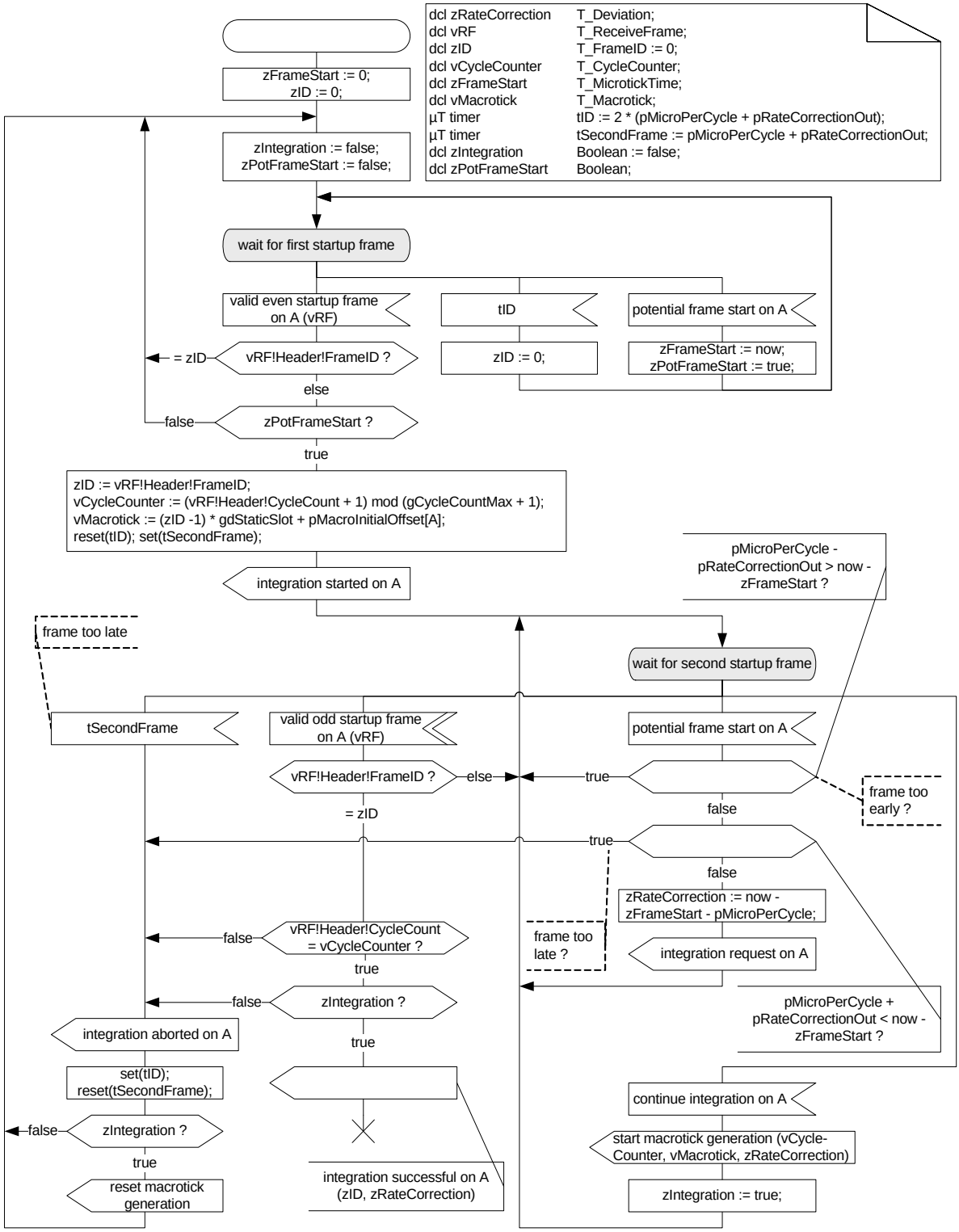

Figure 8-11: Clock synchronization startup process on channel A [CSS_A][112].

[112] The priority input symbol on the *CSS_A:wait for second startup frame* state has been included to resolve the ambiguity that arises if the timer *tSecondFrame* expires at the same time a *valid odd startup frame on A* signal is received.

The reception of the corresponding valid odd startup frame and the satisfaction of the conditions for integration leads to the termination of the CSS process for this channel. Before termination a signal is sent indicating successful integration; this signal causes the INTEGRATION_CONTROL macro of CSP to terminate the CSS process for the other channel (see Figure 8-9). This behavior of this termination is depicted in Figure 8-10.

The timer *tSecondFrame* in Figure 8-11 is used to restart the clock synchronization startup process if the corresponding odd startup frame was not received after an appropriate period of time.

The variable *zID* is used to prohibit attempts to integrate on a coldstart node if an integration attempt on this coldstart node failed in the previous cycle. The timer *tID* prevents this prohibition from applying for more than one double cycle.

8.5 Time measurement

Every node shall measure and store, by channel, the time differences (in microticks) between the expected and the observed arrival times of all sync frames received during the static segment. A data structure is introduced in section 8.5.1. This data structure is used in the explanation of the initialization (section 8.5.2) and the measurement, storage, and deviation calculation mechanisms (section 8.5.3).

8.5.1 Data structure

The following data types are introduced to enable a compact description of mechanisms related to clock synchronization:

```
newtype T_DevValid
struct
      Value    T_Deviation;
      Valid    Boolean;
      Startup  Boolean;
endnewtype;
```

Definition 8-6: Formal definition of T_DevValid.

```
newtype T_ChannelDev
      Array(T_Channel, T_DevValid)
endnewtype;
newtype T_EOChDev
      Array(T_EvenOdd, T_ChannelDev)
endnewtype;
newtype T_DevTable
      Array(T_ArrayIndex, T_EOChDev)
endnewtype;
```

Definition 8-7: Formal definition of T_ChannelDev, T_EOChDev, and T_DevTable.

The structured data type *T_DevTable* is a three dimensional array with the dimensions line number (1 … 15), communication channel (A or B), and communication cycle (even or odd). Each line is used to store the received data of one sync frame pair transmitted by the same node in the same slot in subsequent cycles. If the node is itself a sync node the first line is used to store a deviation of zero, corresponding to the deviation of its own sync frame, for TT-E and TT-L sync nodes two lines have to be reserved. Each element in this three dimensional array contains a deviation value (the structure element Value), a Boolean value indicating whether the deviation value is valid (the structure element Valid), and a Boolean value indicating whether the sync frame corresponding to this deviation was a startup frame (the structure element Startup). Figure 8-12 gives an example of this data structure.

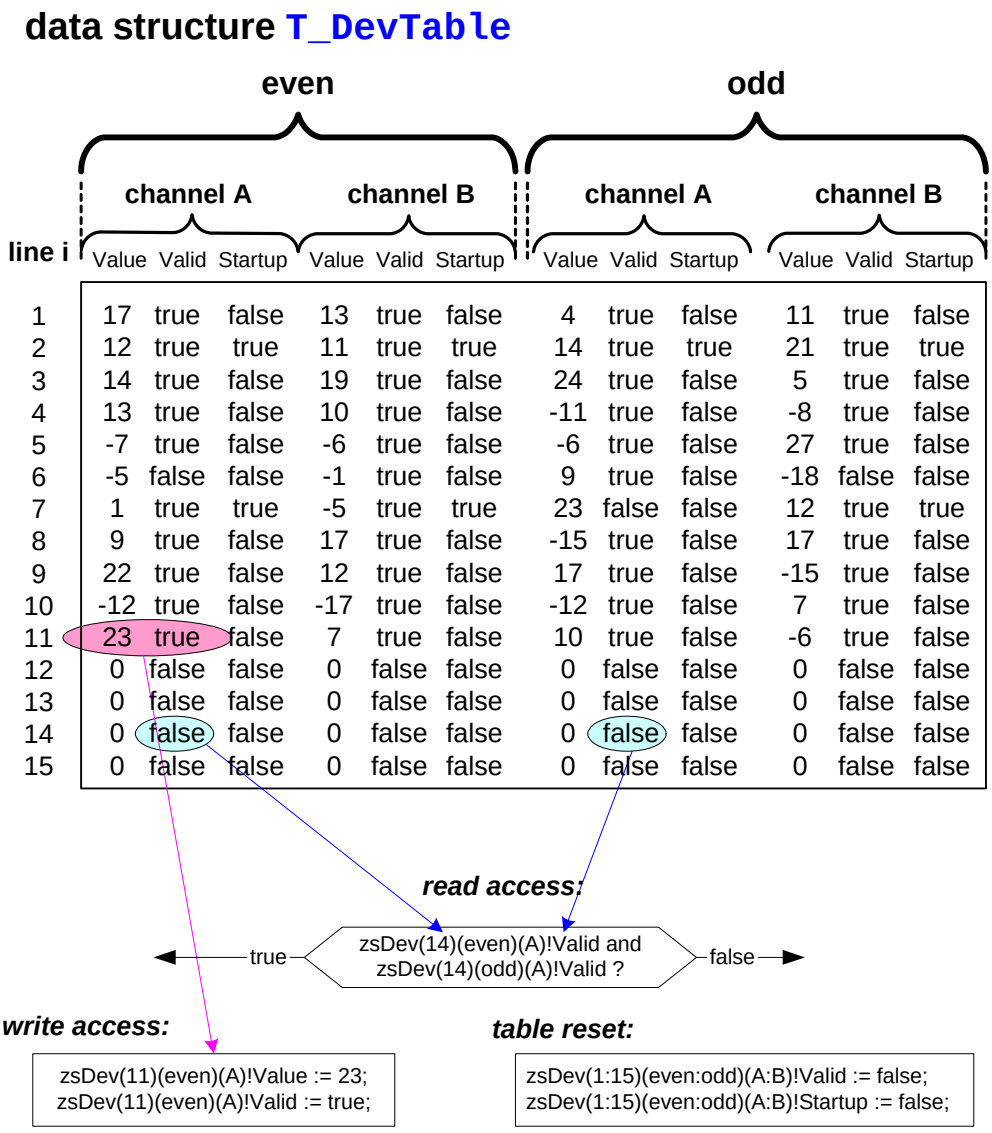

Figure 8-12: Data structure example.

8.5.2 Initialization

The data structure introduced in section 8.5.1 is used to instantiate a variable (*zsDev*). The variable *zsDev* will be initialized at the beginning of an even communication cycle.[113] Additionally, if the node is configured to transmit sync frames (mode SYNC), corresponding entries are stored in the variable as depicted in Figure 8-13. If the node is operating in the two keyslot mode (i.e., is either a TT-L coldstart node or a TT-E coldstart node), entries corresponding to the second transmitted startup frame are also added to the variable *zsDev*. In contrast to the entries for the first sync frame, the entries for the second keyslot are also flagged as belonging to a startup frame, which allows a TT-L coldstart node to proceed through the startup without having received any startup frames from other nodes (see Chapter 7).

[113] TT-E coldstart nodes also initialize *zsDev* on the first cycle of operation regardless of whether it is an even or odd cycle. See Figure 8-5 and Figure 8-7 for details.

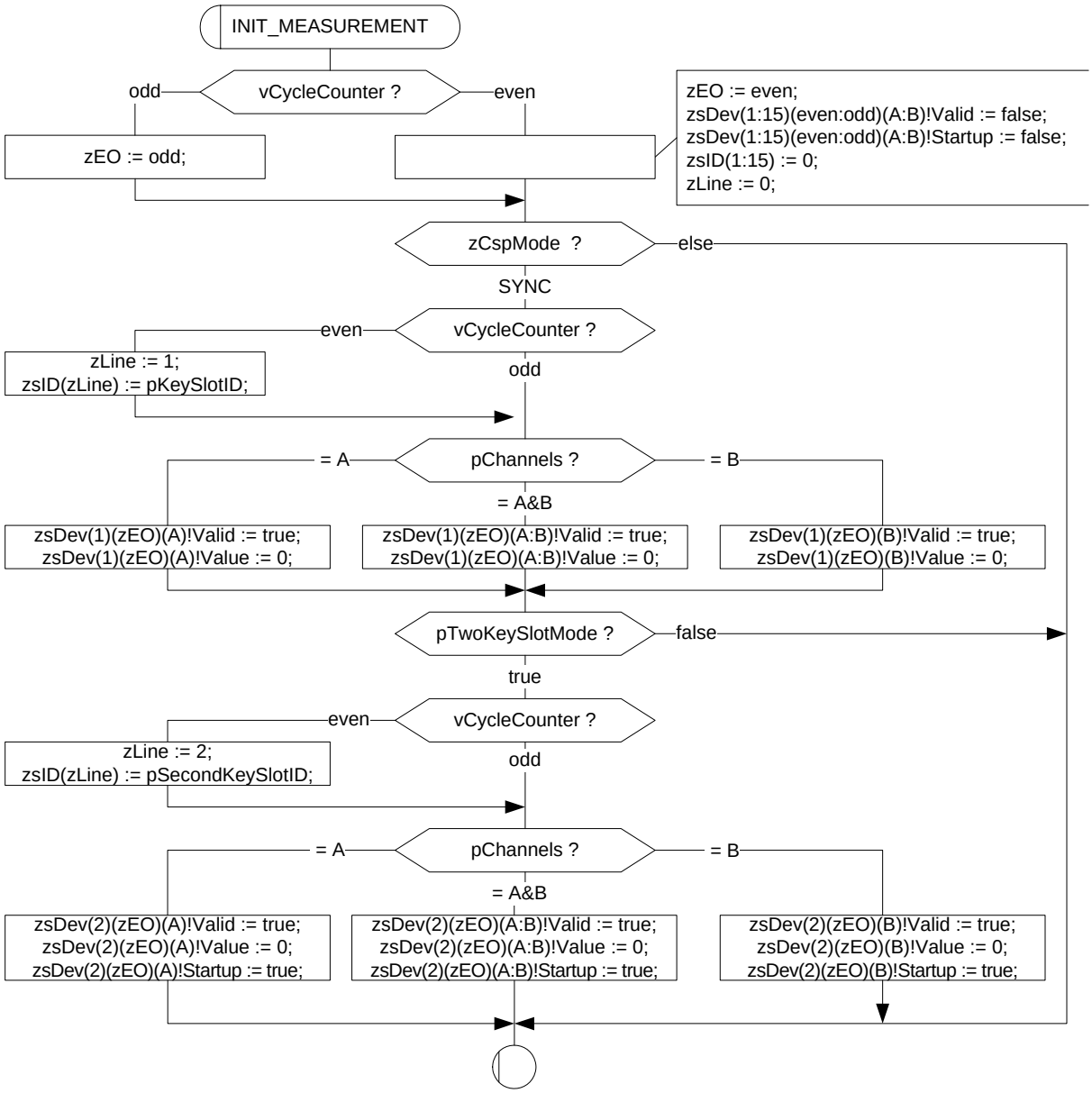

Figure 8-13: Initialization of the data structure for measurement [CSP].

8.5.3 Time measurement storage

The expected arrival time of a static frame is the static slot action point, which is defined in Chapter 5. The MAC generates a signal when the static slot action point is reached. When the clock synchronization process receives this action point signal a time stamp is taken and saved.

During the reception of a frame the decoding unit takes a time stamp when the secondary time reference point is detected. This time stamp is based on the same microtick time base that is used for the static slot action point time stamp. The decoding unit then computes the primary time reference point by subtracting a configurable offset value from the secondary time reference point time stamp. This result is passed to the Frame and Symbol Processing process, which then passes the results to CSP for each valid sync frame received. Further information on the definition of the reference points may be found in section 3.2.6.

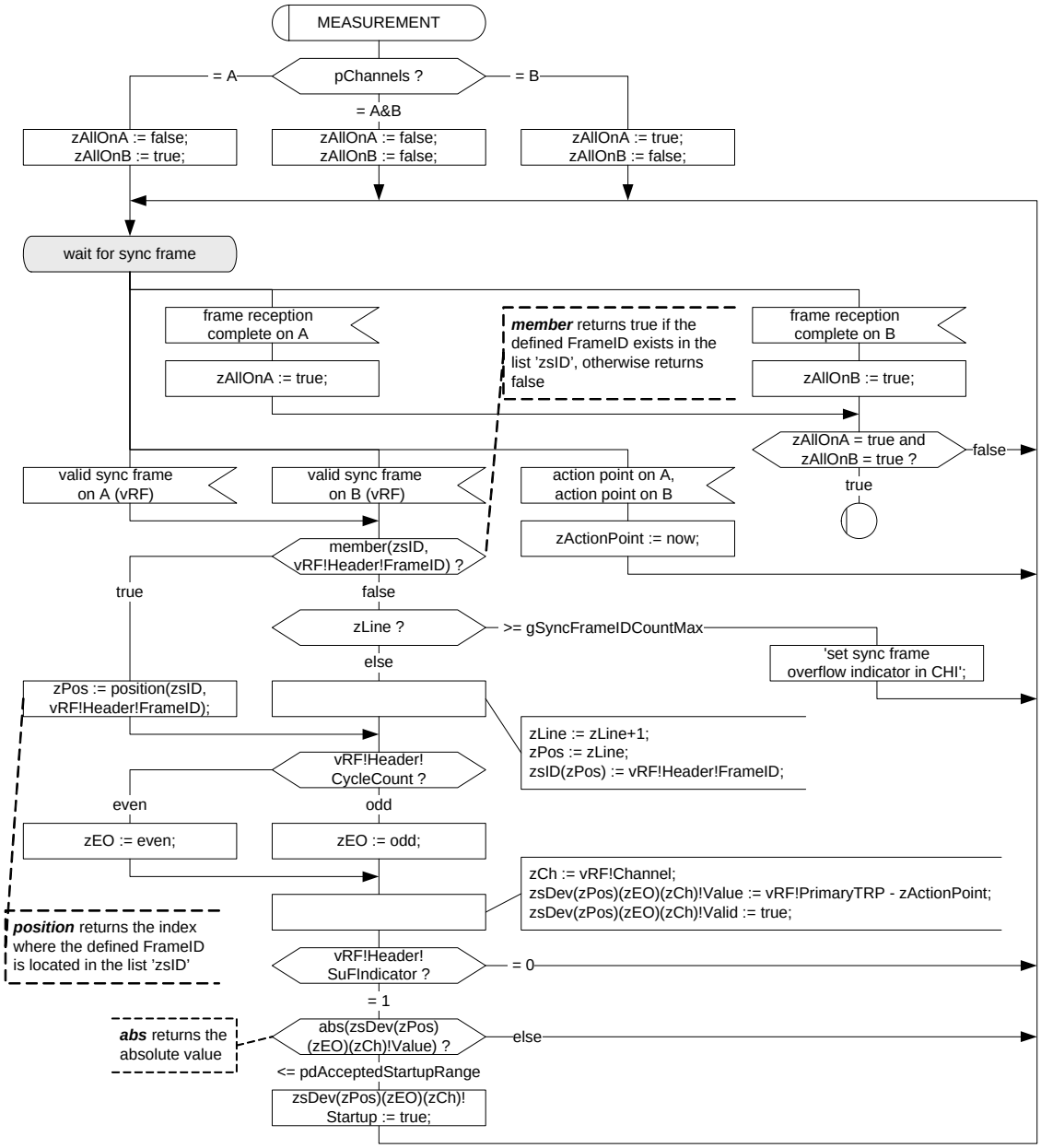

Figure 8-14: Measurement and storage of the deviation values [CSP].

The difference between the action point and primary time reference point time stamps, along with Booleans indicating that the data is valid and whether or not the frame is also a startup frame, is saved in the appropriate location in the previously defined data structure (see Figure 8-14). The measurement phase ends when the static segment ends.

The reception[114] of more than *gSyncFrameIDCountMax* unique sync frame identifiers in either a single communication cycle or an even/odd communication cycle pair indicates an error inside the cluster. This is reported to the host and only measurements corresponding to the first *gSyncFrameIDCountMax* unique sync frame identifiers are considered for the correction value calculations.

8.6 Correction term calculation

8.6.1 Fault-tolerant midpoint algorithm

The technique used for the calculation of the correction terms is a fault-tolerant midpoint algorithm (FTM) (see [Wel88]). The algorithm works as follows (see Figure 8-15 and Figure 8-16):

1. The algorithm determines the value of a parameter, k, based on the number of values in the sorted list (see Table 8-1).[115]

Number of values	k
1 - 2	0
3 - 7	1
> 7	2

Table 8-1: FTM term deletion as a function of list size.

2. The measured values are sorted and the k largest and the k smallest values are discarded.
3. The largest and the smallest of the remaining values are averaged for the calculation of the midpoint value. Note that the division by two of odd numbers should truncate towards zero such that the result is an integer number.[116] The resulting value is assumed to represent the node's deviation from the global time base and serves as the correction term.

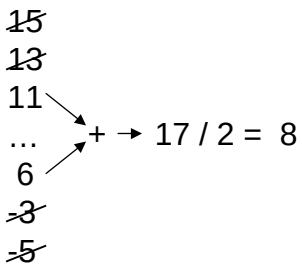

Figure 8-15: Example clock correction value calculation with k=2.

[114] Nodes that transmit sync frames, and are operating in the *POC:normal active* state, will also use their own sync frame measurement values in clock correction. For the purposes of this check, the node's own sync frame is considered to be implicitly received, even though there is no actual reception. As a result, the node's own sync frame is included in the count of sync frame identifiers that is checked against *gSyncFrameIDCountMax*.

[115] The parameter k is not the number of asymmetric faults that can be tolerated.

[116] Example: 17 / 2 = 8 and -17 / 2 = -8.

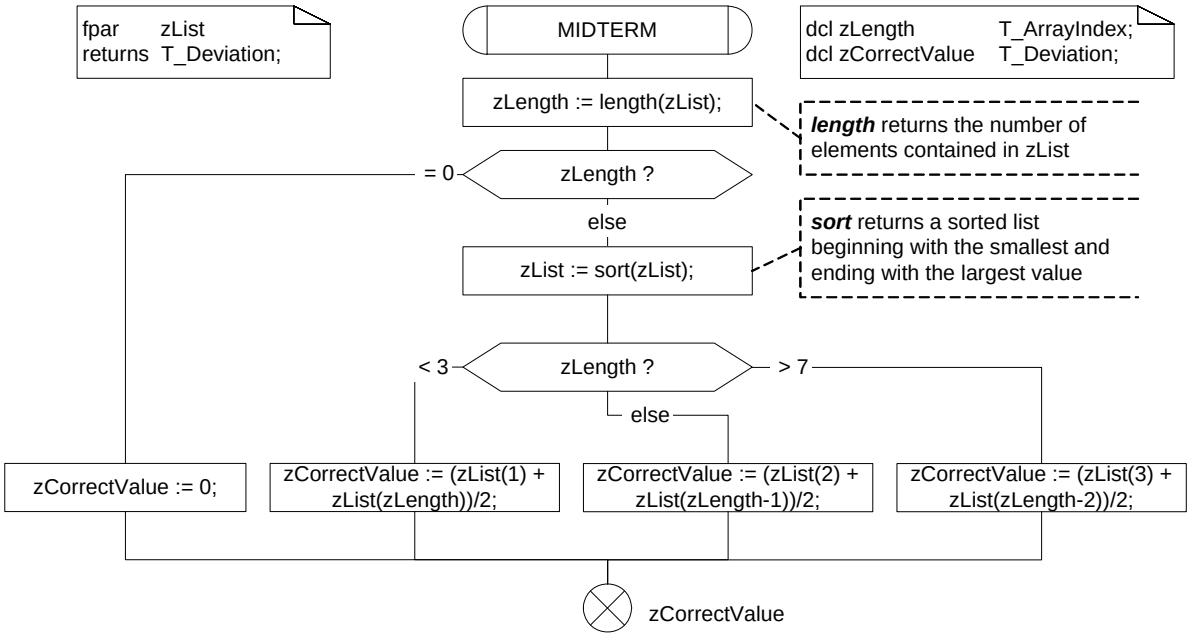

Figure 8-16: Fault-Tolerant Midpoint Procedure [CSP].

8.6.2 Calculation of the offset correction value

The offset correction value *zOffsetCorrection* is a signed integer that indicates how many microticks the node should shift the start of its cycle. Negative values mean the NIT should be shortened (making the next cycle start earlier). Positive values mean the NIT should be lengthened (making the next cycle start later).

In Figure 8-17 the procedure of the calculation of the offset correction value is described in detail. The following steps are covered in the SDL diagram in Figure 8-17:

1. Selection of the previously stored deviation values. Only deviation values that were measured and stored in the current communication cycle are used. If a given sync frame ID has two deviation values (one for channel A and one for channel B) the smaller value will be selected.
2. The number of received sync frames is checked and if an insufficient[117] number of sync frames was received the error condition MISSING_TERM is set.
3. The fault-tolerant midpoint algorithm is executed (see section 8.6.1).
4. The correction term is checked against specified limits. If the correction term is outside of the specified limits the error condition is set to EXCEEDS_BOUNDS and the correction term is set to the maximum or minimum value as appropriate (see section 8.6.4).
5. If appropriate, an external correction value supplied by the host is added to the calculated and checked correction term (see section 8.6.5).

The following data structure is used to save and handle the selected data:

[117] The number of sync frames that need to be received to be considered sufficient depends on the synchronization mode and the node's role. For devices in a TT-D cluster, and for non-coldstart nodes in a TT-E or TT-L cluster, the reception of a single valid sync frame is considered sufficient. TT-E or TT-L coldstart nodes do not need to receive any sync frames, i.e., the number of received sync frames is always considered sufficient. Figure 8-17 implements this behavior not by checking the number of received sync frames directly but rather by checking the number of entries in the *zsMListAB* array.

```
newtype T_DeviationTable
      Array(T_ArrayIndex, T_Deviation)
endnewtype;
```

Definition 8-8: Formal definition of T_DeviationTable.

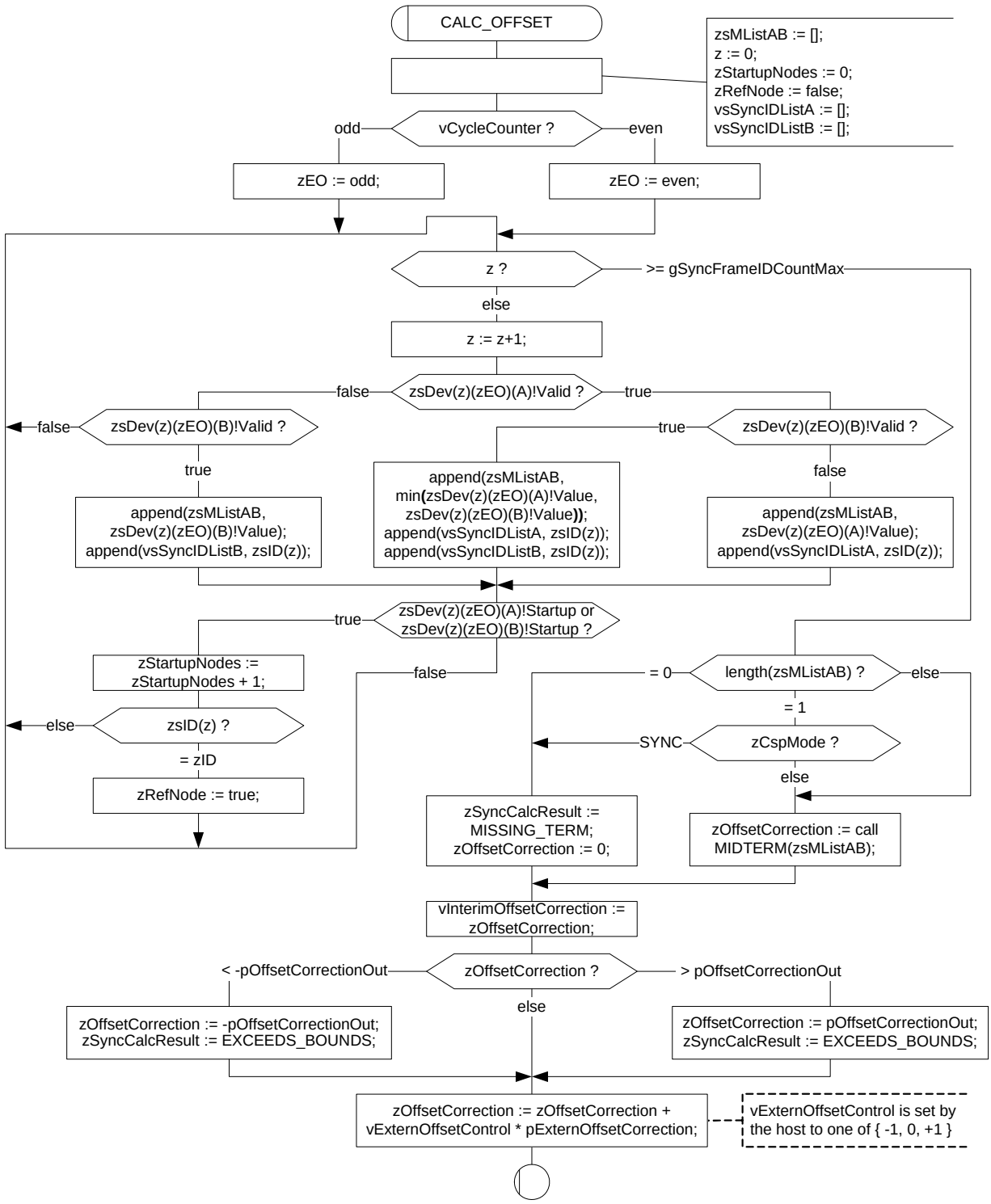

Figure 8-17: Calculation of the offset correction value [CSP].

The SDL abstraction of execution in zero time could lead the reader to the conclusion that the calculation of the offset correction value needs to complete in zero time. This is of course unachievable. It is anticipated that real implementations may take substantial time to calculate the correction, and that implementations may begin the calculation earlier than is shown in Figure 8-6 (i.e., may begin the calculation during the measurement process). Therefore the following restriction on the time required for offset correction calculation is introduced:

The offset correction calculation must be completed no later than *cdMaxOffsetCalculation* after the end of the static segment or 1 MT after the start of the NIT, whichever occurs later.

8.6.3 Calculation of the rate correction value

The goal of the rate correction is to bring the rates of all nodes inside the cluster close together. The rate correction value is determined by comparing the corresponding measured time differences from two successive cycles. A detailed description is given by the SDL diagram depicted in Figure 8-18.

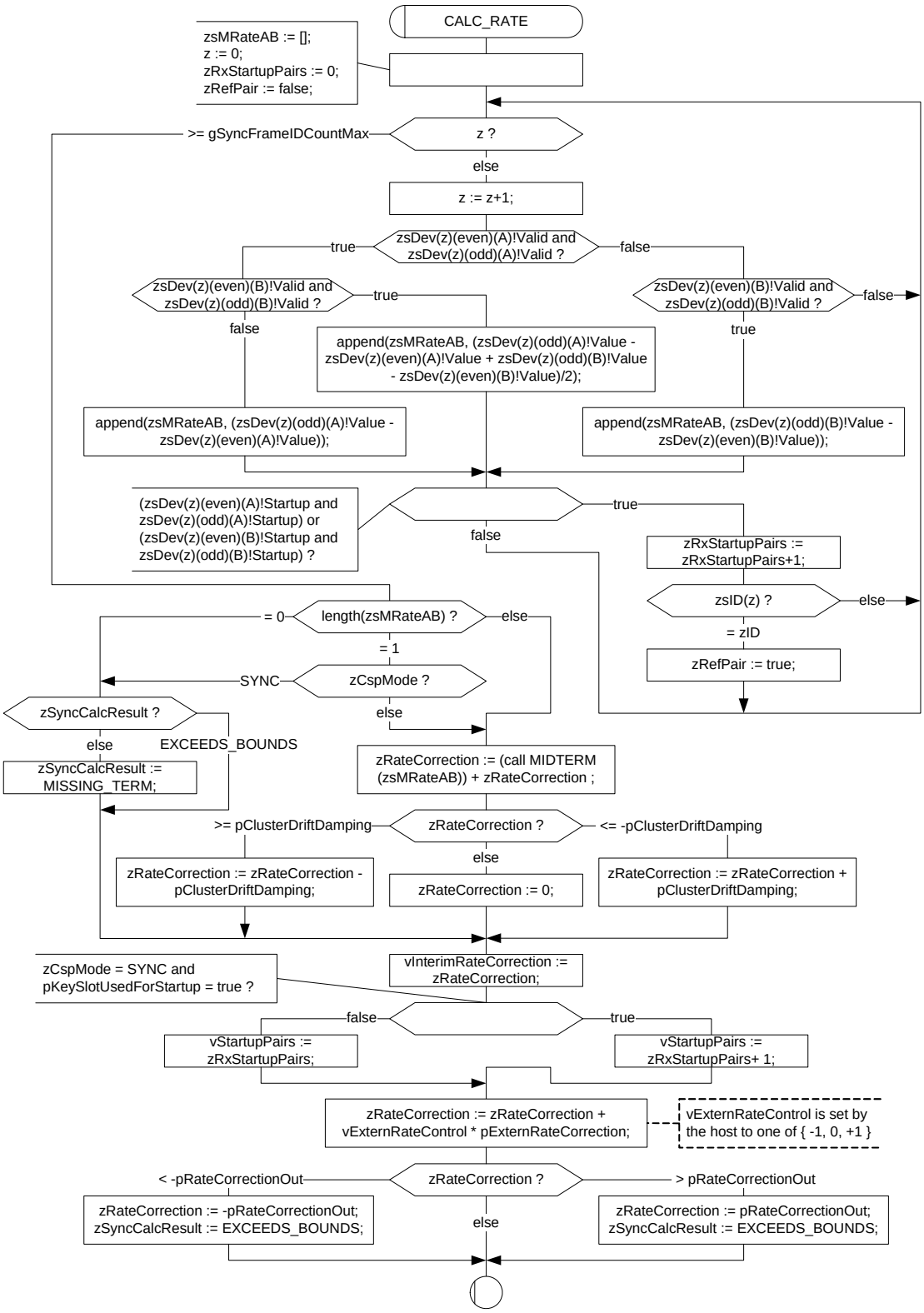

Figure 8-18: Calculation of the rate correction value [CSP].

The rate correction value *zRateCorrection* is a signed integer indicating by how many microticks the node's cycle length should be changed. Negative values mean the cycle should be shortened; positive values mean the cycle should be lengthened.

The following steps are depicted in the SDL diagram in Figure 8-18:

1. Pairs of previously stored deviation values are selected and the difference between the values within a pair is calculated. Pairs are selected that represent sync frames received on the same channel, in slots with the same slot number, on consecutive cycles. If there are two pairs for a given sync frame ID (one pair for channel A and another pair for channel B) the average of the differences is used.
2. The number of received sync frame pairs is checked and if an insufficient[118] number of sync frame pairs was received the error condition MISSING_TERM is set unless the error condition EXCEEDS_BOUNDS had been set previously.[119]
3. The fault-tolerant midpoint algorithm is executed (see section 8.6.1).
4. A damping value *pClusterDriftDamping* for the rate correction term is applied.
5. If appropriate, an external correction value supplied by the host is added to the calculated correction term (see section 8.6.5).
6. The correction term is checked against specified limits. If the correction term exceeds the specified limits the error condition is set to EXCEEDS_BOUNDS and the correction term is set to the maximum or minimum value as appropriate (see section 8.6.4).

In the list above division operations shall produce an integral result that is truncated towards zero.[120]

The *pClusterDriftDamping* parameter should be configured in such a way that the damping value in all nodes has approximately the same duration.[121]

The SDL abstraction of execution in zero time introduces similar problems for the rate correction calculation as are described in section 8.6.2 for the offset correction calculation. Therefore the following restriction on the time required for rate correction calculation is introduced:

> The rate correction calculation must be completed no later than *cdMaxRateCalculation* after the end of the static segment or 2 MT after the start of the NIT, whichever occurs later.

8.6.4 Value limitations

Before applying the calculated correction values, they shall be checked against pre-configured limits (see Figure 8-17 and Figure 8-18).

If correction values are inside the limits, the node is considered fully synchronized.

If either of the correction values is outside of the limits, the node is out of synchronization. This corresponds to an error condition. Information on the handling of this situation is specified in Chapter 2.

The correction values are inside the limits if:

> - *pRateCorrectionOut* <= *zRateCorrection* <= *pRateCorrectionOut*

---

[118] The number of sync frame pairs that need to be received to be considered sufficient depends on the synchronization mode and the node's role. For devices in a TT-D cluster, and for non-coldstart nodes in a TT-E or TT-L cluster, the reception of a single valid sync frame pair is considered sufficient. TT-E or TT-L coldstart nodes do not need to receive any sync frames, i.e., the number of received sync frame pairs is always considered sufficient. Figure 8-18 implements this behavior not by checking the number of received sync frame pairs directly but rather by checking the number of entries in the *zsMRateAB* array.

[119] The consequence of this behavior is that in a TT-D cluster operating with only a single sync node the sync node would detect MISSING_TERM before the non-sync nodes. The non-sync nodes would not detect MISSING_TERM until the last sync node has stopped transmitting, presumably due to loss of synchronization. As a consequence, in this circumstance different types of nodes would cease operation at different times.

[120] Example: 17 / 2 = 8 and -17 / 2 = -8.

[121] A node-specific configuration value is used to allow clusters that have different microtick durations in different nodes.

- *pOffsetCorrectionOut* <= *zOffsetCorrection* <= *pOffsetCorrectionOut*

If both correction values are inside the limits the correction will be performed; if either value exceeds its pre-configured limit, an error is reported and the node enters the *POC:normal passive* or the *POC:halt* state depending on the configured behavior (see Chapter 2). If a value is outside its pre-configured limit it is reduced or increased to its limit. If operation continues, the correction is performed with this modified value.

8.6.5 Host-controlled external clock synchronization

During normal operation independent clusters can drift significantly. If synchronous operation is desired across multiple clusters, external synchronization is necessary even though the nodes within each cluster are synchronized. This can be accomplished by the synchronous application of host-controlled external rate and offset correction terms to both clusters.

The control data *vExternRateControl* and *vExternOffsetControl* of the external clock correction have three different values, +1 / -1 / 0, with the following meanings:

Value:	**+1**	**-1**	**0**
rate correction *vExternRateControl*	increase cycle length by *pExternRateCorrection*	decrease cycle length by *pExternRateCorrection*	no change
offset correction *vExternOffsetControl*	start cycle later by *pExternOffsetCorrection*	start cycle earlier by *pExternOffsetCorrection*	no change

Table 8-2: External clock correction control.

The size of the external rate and the external offset correction values *pExternOffsetCorrection* and *pExtern-RateCorrection* are fixed and configured in the *POC:config* state.

The application must ensure that the external offset correction is performed in the same cycle with the same value in all nodes of a cluster and that the external rate correction is performed in the same double cycle with the same value in all nodes of a cluster.

The type is defined as follows:

```
syntype T_ExternCorrection = Integer
      constants -1, 0, +1
endsyntype;
```

Definition 8-9: Formal definition of T_ExternCorrection.

The handling of the external correction values is depicted in Figure 8-17 and Figure 8-18.

The configuration must ensure that the addition of the external correction value can be performed even if the pre-configured limit is exceeded by the addition of an external correction term.

8.6.6 TT-E time gateway sink correction determination

A node operating as TT-E coldstart node supports additional functionality which is described in Figures 8-19 to 8-24. This functionality is optional and is shown in the SDL diagrams as a dashed outline box labeled "TT-E time gateway sink behavior".

A TT-E coldstart node calculates the clock correction values as any other node in the cluster based on the measured deviation between the expected and the observed arrival times of the sync frames, even though the calculated clock correction values will eventually be overwritten by the clock correction values provided by the time gateway source node.

If a TT-E coldstart node is configured to switch to the local synchronization method in case the synchronization with the time gateway source node is lost, it will use the clock correction values it has calculated itself. However, as a TT-E coldstart node configured to switch to the local synchronization method must necessarily be the sole coldstart node of the TT-E cluster, the calculated correction values will always be zero, just as they are for a TT-L coldstart node.

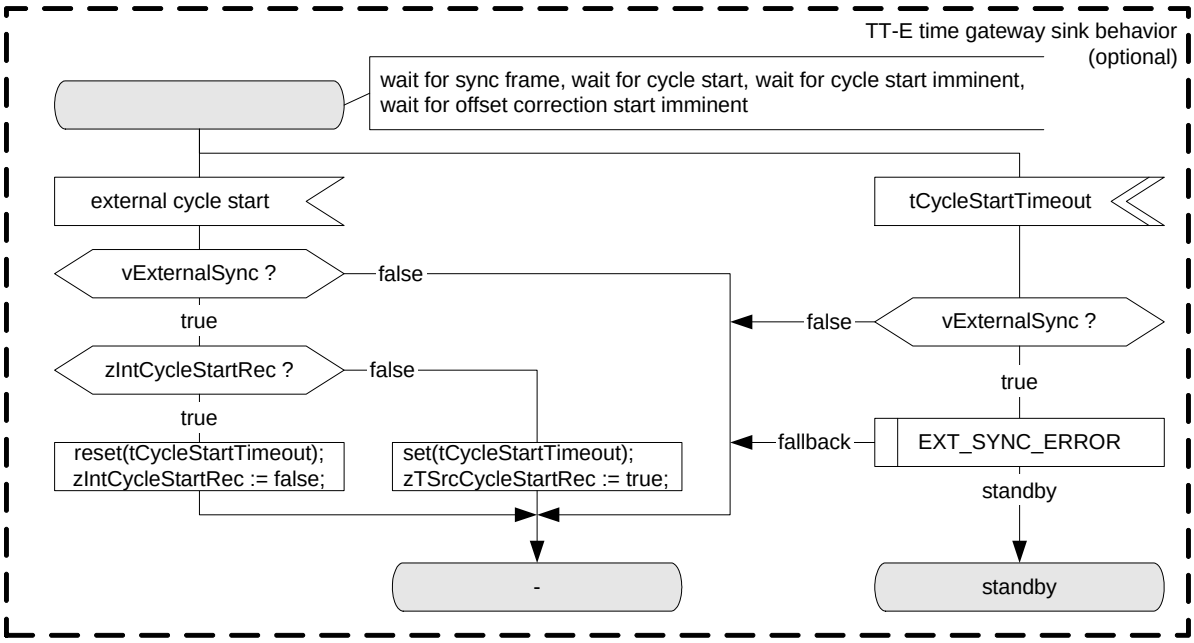

Figure 8-19: Cycle start supervision [CSP].

Figure 8-19 above describes how the time gateway sink node continuously monitors that its locally generated *cycle start* signal is still synchronous to the one periodically sent by its time gateway source node. Should the externally generated cycle start event and the locally generated cycle start event be *cdCycleStartTimeout* or more microticks apart, the time gateway sink node assumes that it has lost synchronization to the time gateway source node (see Figure 8-24).

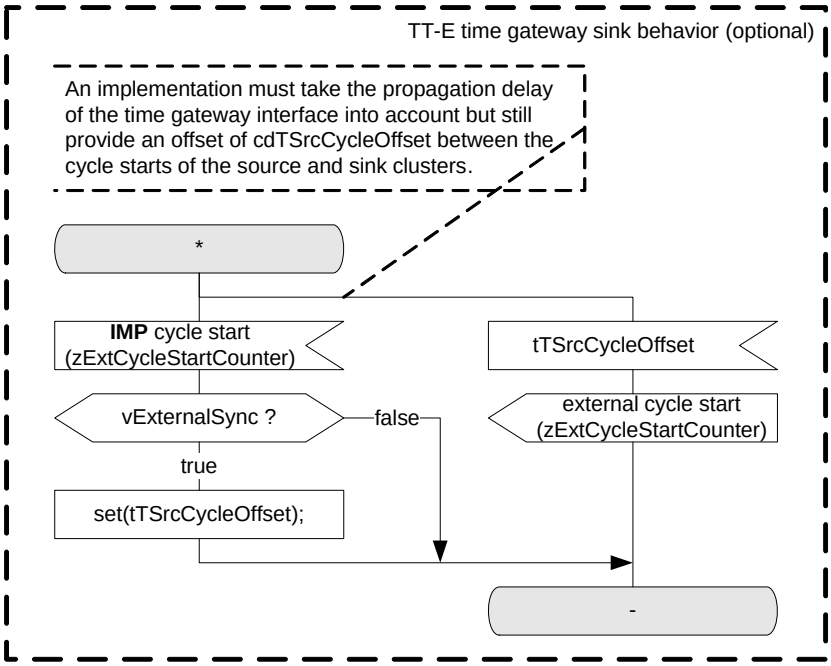

Figure 8-20: External cycle start delay [CSP].

The delay timer *tTSrcCycleOffset* provides a delay between the cycle starts of the time gateway source and time gateway sink clusters. It is intended that this offset is a protocol constant (i.e., *cdTSrcCycleOffset*). The SDL description makes the assumption that the cycle start event propagates across the time gateway interface in zero time, and thus the SDL describes a delay that is equal to the delay that is required between the two cycle starts. In practice, propagation across the time gateway interface will not take place in zero time – an implementation must take this propagation time into account but still provide an offset of *cdTSrc-CycleOffset* between the cycle starts in the two clusters. Note that this may require an implementation to provide one or more configuration parameters, not described in this specification, if the propagation delay across the time gateway interface is not under the control of the implementation (for example, if the time gateway interface is external to the implementation).

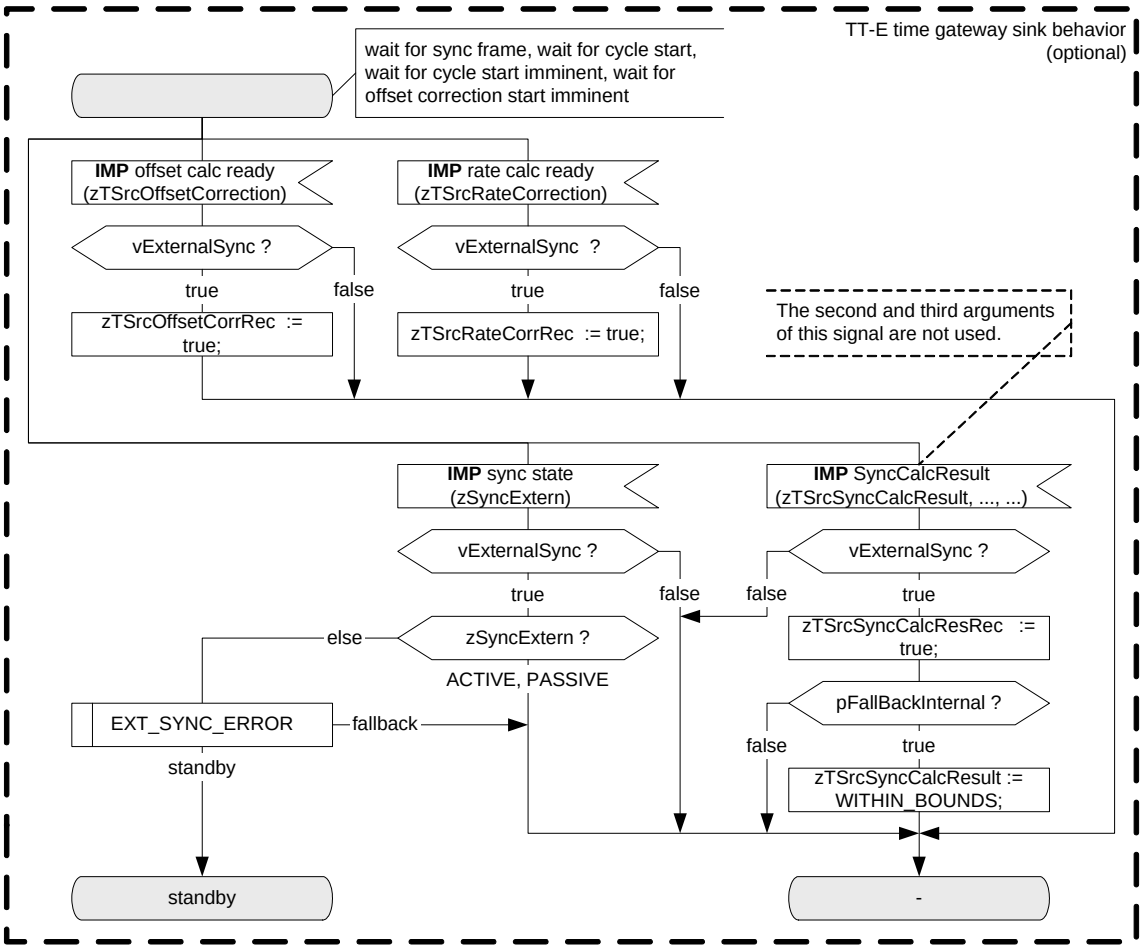

Figure 8-21: Obtain external clock sync signals [CSP].

The time gateway sink node continuously monitors the time gateway interface and stores any received updates of clock correction values as depicted in Figure 8-21. Should the time gateway source node indicate that it is neither in the state *POC:normal active* nor in the state *POC:normal passive*, the time gateway sink node will behave as if it has lost synchronization to the time gateway source node (see Figure 8-24).

If the time gateway sink node is the sole provider of startup frames of the cluster, it must not switch to *POC:normal passive*. To that end, the node overwrites the *zSyncCalcResult* value of the time gateway source node with the fault free value "WITHIN_BOUNDS" if it is configured to "fall back" to the local synchronization method, which implies that it is the sole coldstart node. If several TT-E coldstart nodes are present in the cluster, it is obviously important that they all agree upon the schedule. Should the time gateway source node of one of the time gateway sink nodes have a problem with its clock synchronization, it cannot ascertain the validity of its view on the time source cluster schedule. Therefore, its time gateway sink node should (also) enter *POC:normal passive* and only return to *POC:normal active* when its time gateway source node has recovered its synchronization with the time source cluster.

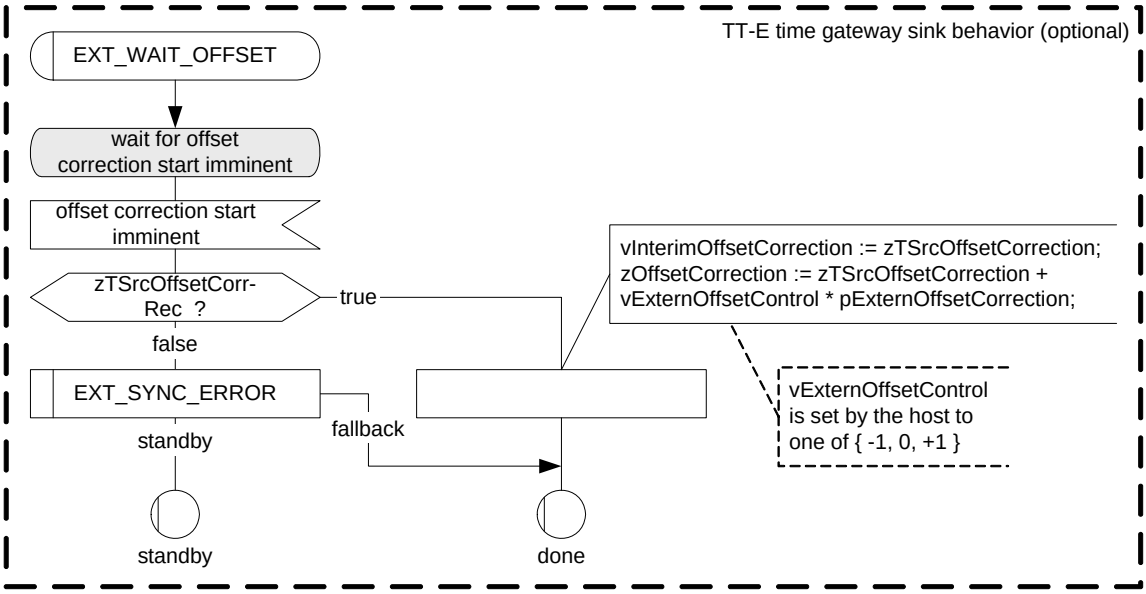

Figure 8-22: Macro EXT_WAIT_OFFSET [CSP].

The time gateway sink node requires an update of the offset correction term before it enters the offset correction phase. Figure 8-22 shows how it checks just before the start of the offset correction phase whether it has received a new offset correction term from its time gateway source node. If it has not received an update, it will behave as if it has lost synchronization to the time gateway source node (see Figure 8-24). If it has received an update, it will discard its locally generated offset correction term and use the one supplied by the time gateway source node instead.

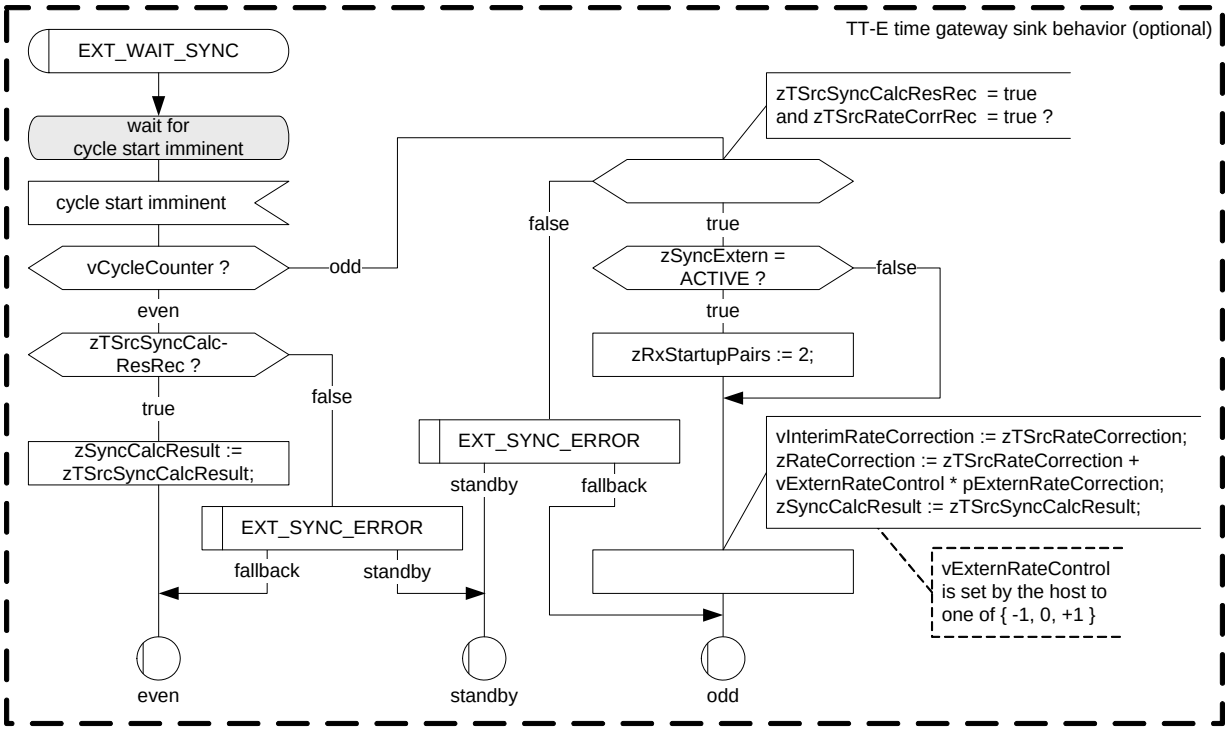

Figure 8-23: Macro EXT_WAIT_SYNC [CSP].

The time gateway sink node requires an update of the rate correction term before it enters a new even cycle. Figure 8-23 shows how it checks just before the start of the new even cycle whether it was provided with a new rate correction term by its time gateway source node. If it has not received such an update, it will behave as if it has lost synchronization to the time gateway source node (see Figure 8-24).

If the time gateway sink node has received an update, it will discard its locally generated rate correction term and use the one supplied by the time gateway source node instead. In addition, if the *zSyncExtern* variable indicates that the time gateway source node is in *POC:normal active* the time gateway sink will set the variable *zRxStartupPairs* to 2 - this allows the time gateway sink to follow the time gateway source as it makes a transition from *POC:normal passive* to *POC:normal active*.

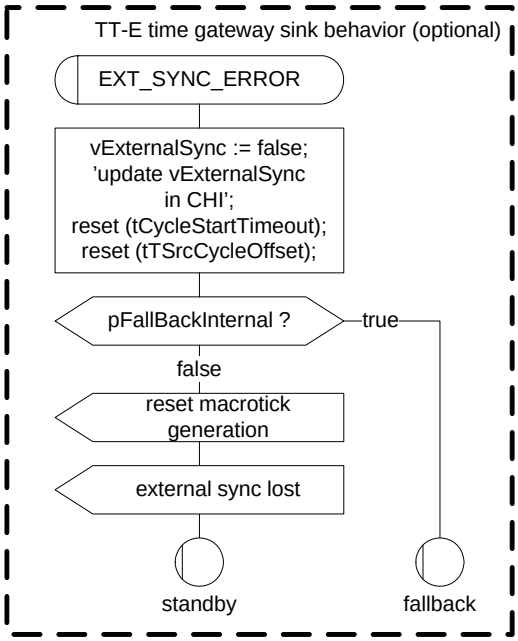

Figure 8-24: Macro EXT_SYNC_ERROR [CSP].

The macro EXT_SYNC_ERROR describes the behavior of the time gateway sink node in the case that it determines a loss of synchronization with the time gateway source node. In such a case, the time gateway sink node first informs the CHI. The CSP then checks whether it is configured to "fall back" to the local synchronization method. If yes, it does so; if not, it stops the macrotick generation process and indicates the loss of synchronization to the POC, which will then enter *POC:halt*.

8.7 Clock correction

Once calculated, the correction terms are used to modify the local clock in a manner that synchronizes it more closely with the global clock. This is accomplished by using the correction terms to adjust the number of microticks in each macrotick.

The macrotick generation (MTG) process (Figure 8-25) generates (corrected) macroticks. There are three different ways to begin the process of generating macroticks:

- For a leading coldstart node, the protocol operation control (POC) process initiates macrotick generation (via the *coldstart* signal) if the conditions to start the node as a coldstart node are satisfied, or
- For an integrating or following coldstart node, the clock synchronization startup (CSS) processes initiate macrotick generation (via the *start macrotick generation* signal) upon the reception of an acceptable pair of startup frames, or
- For a TT-E coldstart node, the clock synchronization process (CSP) initiates macrotick generation (via the *start macrotick generation* signal) if the conditions to perform the startup of the time sink cluster are satisfied.

Either of the two paths will set initial values for the cycle counter, the macrotick counter, and the rate correction value. A loop will be executed every microtick and, as a result, macroticks are generated that include a uniform distribution of a correction term over the entire time range. This loop is only left if the macrotick generation process is terminated by the POC process (e.g. in case of an error) or if a *reset macrotick generation* signal is received from the POC, CSP, CSS_A, or CSS_B process.

The relevant time range for the application of the rate correction term is the entire cycle; the time range for the application of the offset correction term is the time between the start of the offset correction until the next cycle start. The macrotick generation process handles this by two different initializations. At the cycle start the algorithm is initialized using only the rate correction value; at the start of the offset correction phase the algorithm is initialized again, this time including the offset correction values.

Concurrent with the MTG process new measurement values are taken by the CSP and these values are used to calculate new correction values. These new correction values are ultimately applied and used by the macrotick generation process. The new offset correction value is applied at the start of offset correction period in an odd communication cycle, and the new rate correction value is applied at the cycle start in an even communication cycle.

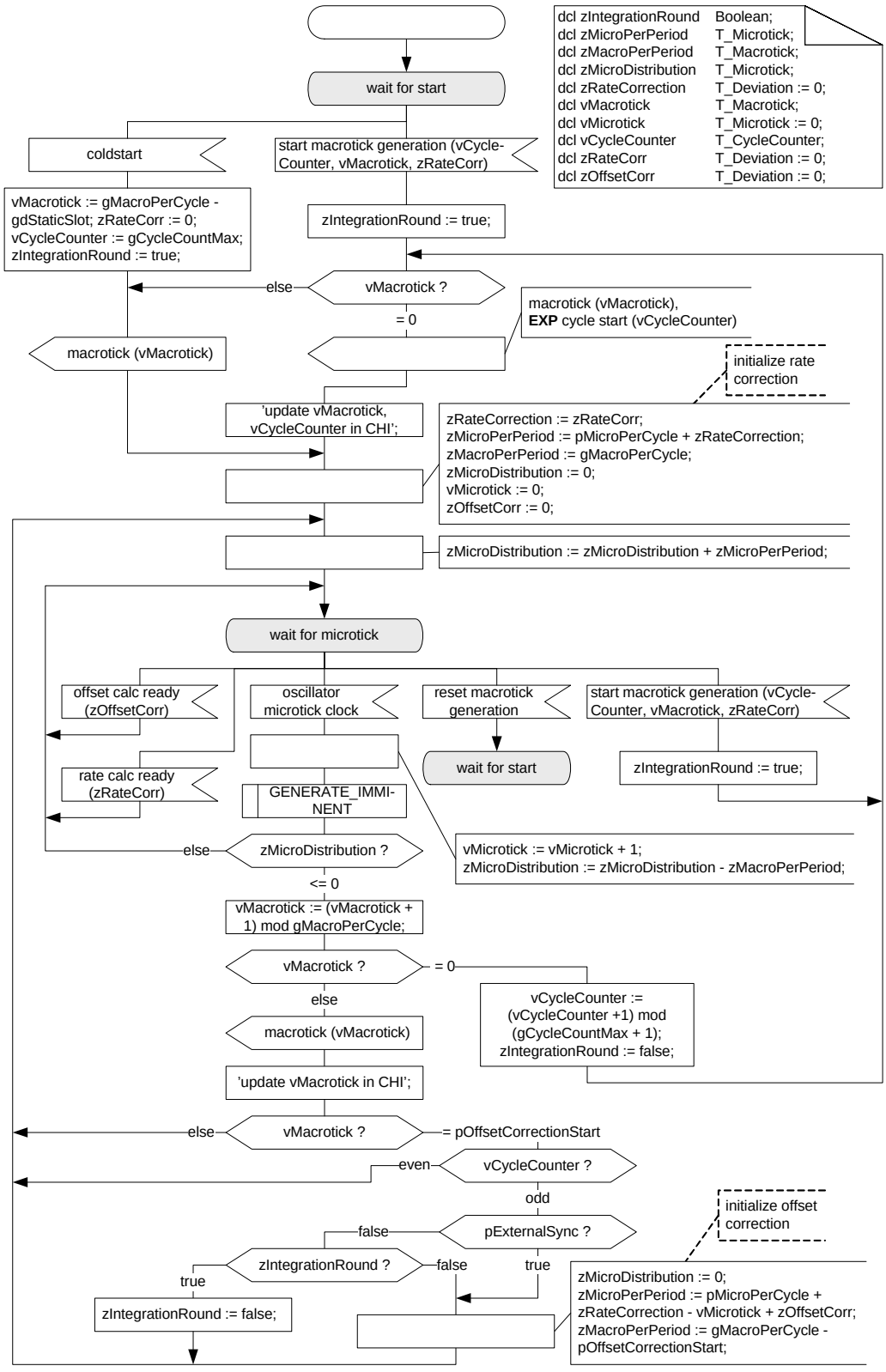

Figure 8-25: Macrotick generation [MTG].

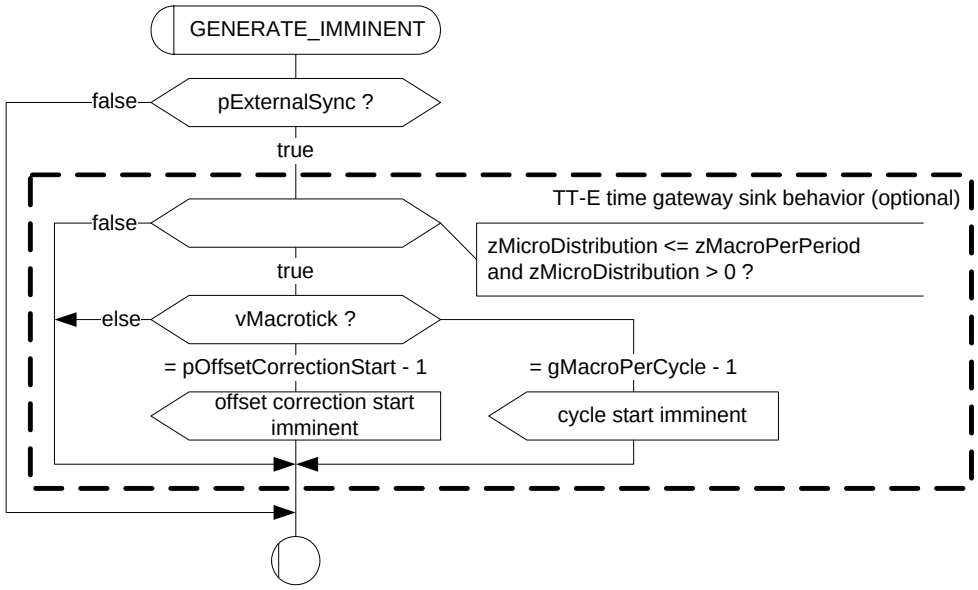

Figure 8-26: Macro GENERATE_IMMINENT [MTG].

The macro GENERATE_IMMINENT encapsulates the generation of two signals used only by TT-E coldstart nodes. These signals are used to determine the points in time at which a TT-E coldstart node decides that it has lost synchronization with its time gateway source if it has not received the appropriate clock correction value. The offset correction value should be received before the *offset correction start imminent* signal and the rate correction value should be received before the *cycle start imminent* signal, otherwise the TT-E coldstart node will assume a loss of synchronization.

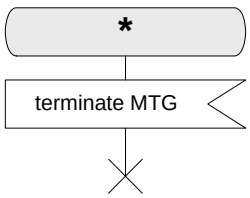

Figure 8-27: Termination of the MTG process [MTG].

8.8 Sync frame configuration rules

FlexRay supports a distributed clock synchronization that can be configured in many different ways. Table 8-3 illustrates a number of rules constraining the possible configurations.

Element	Limit, range		
	TT-D	TT-L	TT-E
Number of sync nodes	2^a … *cSyncFrameIDCountMax*	1	1 … **floor(** *cSyncFrameIDCountMax* / 2 **)**

Table 8-3: Configuration rules for the clock synchronization.

Number of sync frame ID's	= number of sync nodes	2	= 2 * number of sync nodes
Number of static slots	>= number of sync frame ID's	>= 2	>= number of sync frame ID's

Table 8-3: Configuration rules for the clock synchronization.

a. A TT-D cluster with only two sync nodes (each of which is also configured to be a coldstart node) would be able to start up and continue to operate, but a loss of either of the two sync nodes would result in the loss of all communication.

8.8.1 TT-D cluster

A TT-D cluster consists of a number of sync nodes and an arbitrary number of non-sync nodes. Each coldstart node is always a sync node but there can be more sync nodes than coldstart nodes. A number of nodes must be configured as sync nodes depending on the following rules:

- At least two nodes shall be configured to be sync nodes.
- At least two of the sync nodes must also be TT-D coldstart nodes.
- At most *cSyncFrameIDCountMax* nodes shall be configured to be sync nodes.
- Only nodes with *pChannels* = *gChannels* may be sync nodes (i.e., sync nodes must be connected to all configured channels).
- Sync nodes that support two channels shall send two sync frames in the static segment, one on each channel in the corresponding slot with the slot number which equals *pKeySlotID*.[122] Sync nodes that only support a single channel shall send one sync frame in the static segment. The sync frame shall be sent in the slot with the slot number which equals *pKeySlotID*.
- The sync frames shall be sent in all slots with the same slot number, i.e. in each cycle.
- Non-sync nodes must not transmit frames with the sync frame indicator set to one.

8.8.2 TT-E cluster

A TT-E cluster consists only of coldstart nodes and an arbitrary number of non-sync nodes. In that respect the following rules must be observed:

- At least one node shall be configured to be a sync node.
- Each sync node must also be a TT-E coldstart node.
- At most *cSyncFrameIDCountMax* / 2 nodes shall be configured to be sync nodes.
- Only nodes with *pChannels* = *gChannels* may be sync nodes (i.e., sync nodes must be connected to all configured channels).
- Sync nodes that support two channels shall send four sync frames in the static segment, two on each channel, in the slots with slot number equal to *pKeySlotID* or *pSecondKeySlotID*.[123] Sync nodes that support only a single channel shall send two sync frames in the static segment in the slots with slot number equal to *pKeySlotID* or *pSecondKeySlotID*.
- The sync frames shall be sent in all slots with the same slot number, i.e. in each cycle.
- Non-sync nodes must not transmit frames with the sync frame indicator set to one.

8.8.3 TT-L cluster

A TT-L cluster consists only of a single coldstart node and an arbitrary number of non-sync nodes. In that respect the following rules must be observed:

[122] The frames sent on the two channels in these slots do not need to be identical (i.e., they may have differing payloads), but they must both be sync frames.

[123] The frames sent in these slots by a TT-E sync node do not need to be identical (i.e., different payloads may be sent on each channel, or in each of the key slots), but all must be sync frames.

- A single node shall be configured to be a sync node.
- The single sync node must be a TT-L coldstart node.
- Only nodes with *pChannels* = *gChannels* may be sync nodes (i.e., sync nodes must be connected to all configured channels).
- Sync nodes that support two channels shall send four sync frames in the static segment, two on each channel, in the slots with slot number equal to *pKeySlotID* or *pSecondKeySlotID*.[124] Sync nodes that support only a single channel shall send two sync frames in the static segment in the slots with slot number equal to *pKeySlotID* or *pSecondKeySlotID*.
- The sync frames shall be sent in all slots with the same slot number, i.e. in each cycle.
- Non-sync nodes must not transmit frames with the sync frame indicator set to one.

8.9 Time gateway interface

The time gateway interface connects two communication controllers. It provides information about the schedule of the time gateway source node to the time gateway sink node.

The signals to be provided by the time gateway source node are marked in the SDL description using the EXP keyword. The signals in question are

- *cycle start*
  This signal is used by the time gateway sink node to initialize its own schedule and later on to monitor if it still operates synchronously to the time gateway source node.
- *sync state*
  This signal is used by the time gateway sink node to determine if the time gateway source node is in *POC:normal active*, *POC:normal passive* or a different state.
- *offset calc ready*
  This signal is used by the time gateway sink node to perform the same clock offset correction as the time gateway source node.
- *rate calc ready*
  This signal is used by the time gateway sink node to perform the same clock rate correction as the time gateway source node.
- *SyncCalcResult*
  This signal is used by the time gateway sink node to determine if the time gateway source node maintains synchronization with the time source cluster.

The fixed offset of *cdTSrcCycleOffset* microticks between the schedule of the time gateway source node and the time gateway sink node ensures that each transferred piece of information arrives in time to be properly used by the time gateway sink node.

[124] The frames sent in these slots by a TT-L sync node do not need to be identical (i.e., different payloads may be sent on each channel, or in each of the key slots), but all must be sync frames.

Chapter 9
Controller Host Interface

9.1 Principles

The controller host interface (CHI) manages the data and control flow between the host processor and the FlexRay protocol engine within each node.[125]

The CHI contains two major interface blocks - the protocol data interface and the message data interface. The protocol data interface manages all data exchange relevant for the protocol operation and the message data interface manages all data exchange relevant for the exchange of messages as illustrated in Figure 9-1.

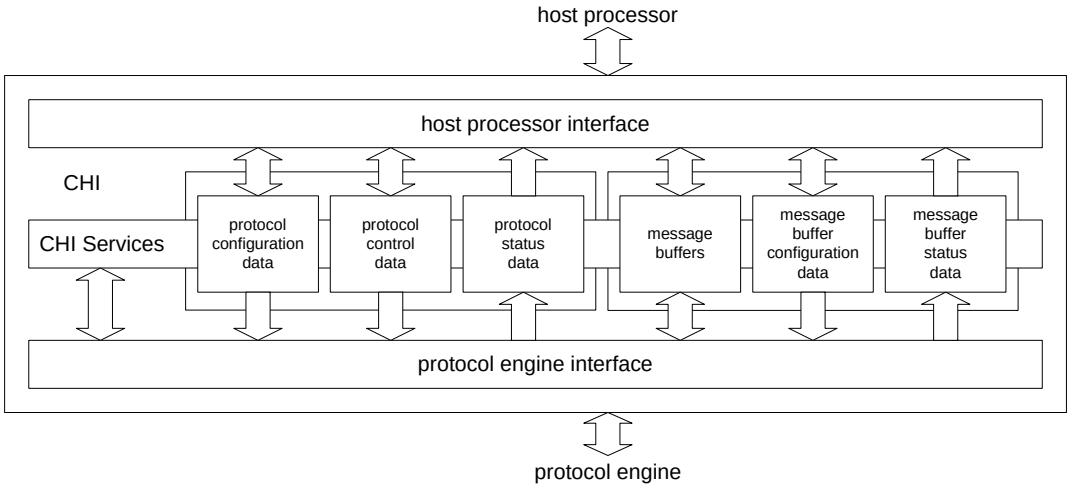

Figure 9-1: Conceptual architecture of the controller host interface.

The protocol data interface manages the protocol configuration data, the protocol control data, and the protocol status data. The message data interface manages the message buffers, the message buffer configuration data, and the message buffer status data.

In addition, the CHI provides a set of CHI services that define self-contained functionality that is transparent to the operation of the protocol.

The descriptions of the CHI in this chapter are behavioral descriptions, not requirements on a particular method of implementation. In many cases the method of description was chosen for ease of understanding rather than efficiency of implementation. An actual implementation should have the same behavior as the description, but it need not have the same underlying structure or mechanisms.

9.2 Description

The relationships between the CHI and the other protocol processes are depicted in Figure 9-2[126].

[125] Please note that due to implementation constraints the CHI may add product specific delays for data or control signals exchanged between the host and the protocol engine.

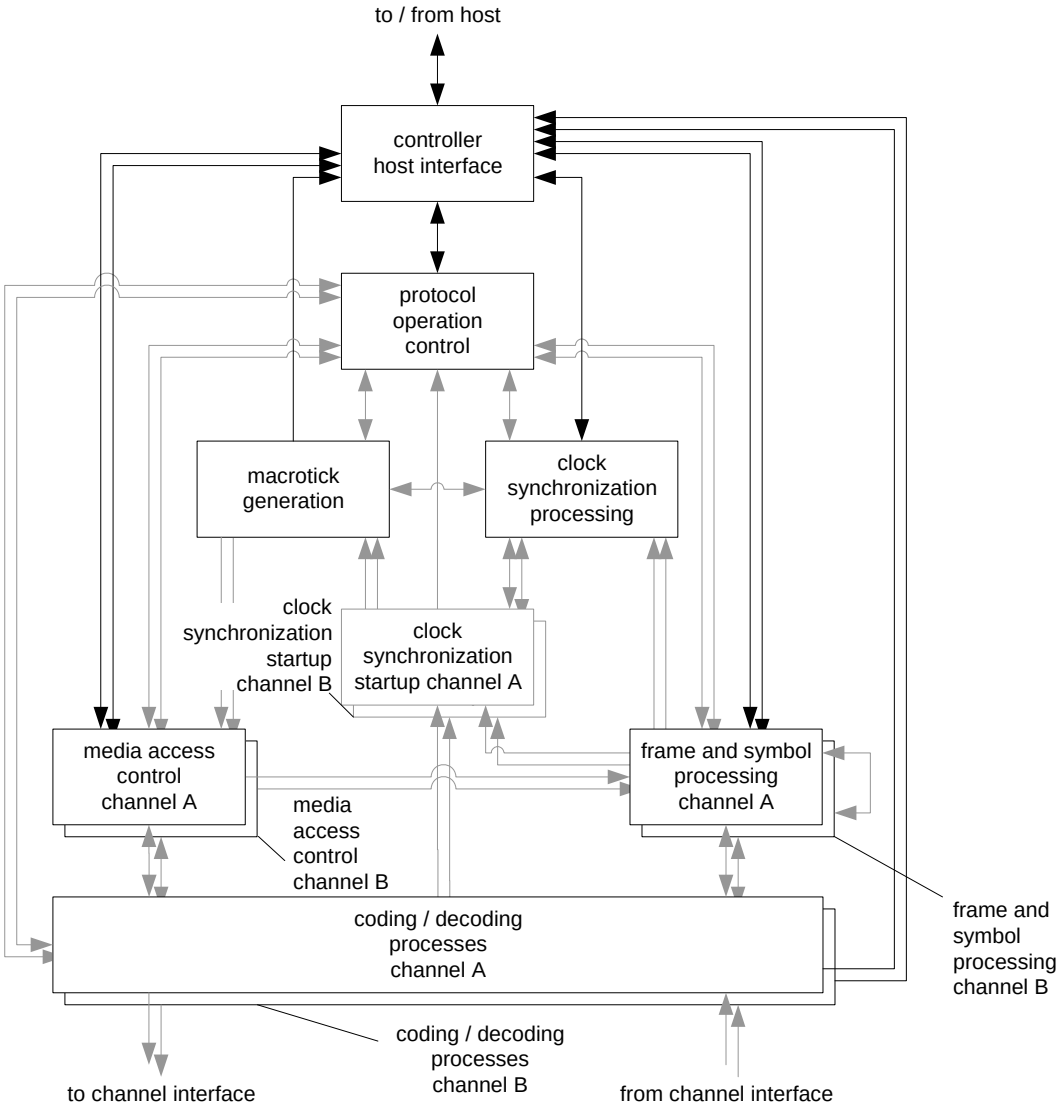

Figure 9-2: Controller host interface context.

9.3 Interfaces

9.3.1 Protocol data interface

9.3.1.1 Protocol configuration data

The host shall have write access to protocol configuration data only when the protocol is in the *POC:config* state.

The host shall have read access to protocol configuration data regardless of the protocol state.

[126] The dark lines represent data flows between mechanisms that are relevant to this chapter. The lighter gray lines are relevant to the protocol, but not to this chapter.

9.3.1.1.1 Communication cycle timing configuration

All configuration data relating to the following communication cycle timing parameters shall be treated as protocol configuration data:

1. the number of microticks *pMicroPerCycle* constituting the duration of the communication cycle,
2. the number of macroticks *gMacroPerCycle* within a communication cycle,
3. the number of static slots *gNumberOfStaticSlots* in the static segment,
4. the number of macroticks *gdStaticSlot* constituting the duration of a static slot within the static segment,
5. the number of macroticks *gdActionPointOffset* constituting the offset of the action point within static slots,
6. the number of macroticks *gdMinislot* constituting the duration of a minislot,
7. the number of minislots *gNumberOfMinislots* within the dynamic segment,
8. the number of macroticks *gdMinislotActionPointOffset* constituting the offset of the action point within a minislot of the dynamic slot,
9. the number of minislots *gdDynamicSlotIdlePhase* constituting the duration of the idle phase within a dynamic slot,
10. the number of the last minislot *pLatestTx* in which transmission can start in the dynamic segment,
11. the number of macroticks *gdSymbolWindow* constituting the duration of the symbol window,
12. the number of macroticks *gdSymbolWindowActionPointOffset* constituting the offset of the action point within the symbol window,
13. the cycle number *gCycleCountMax* after which the cycle counter is reset back to zero[127].

In addition to the above, an implementation shall allow a system designer control over the parameters *gdSampleClockPeriod* and *pSamplesPerMicrotick*. These parameters need not be directly implemented, but may instead be derived from other parameters not explicitly defined in this specification. For example, *gdSampleClockPeriod* may be derived from parameters that set the prescalers, phase-locked loop multipliers, etc. for an implementation's sample clock and the designer's knowledge of the design frequency of the underlying clocks. As a result, there is no specific requirement to be able to read or configure these exact parameters, but there is a requirement for a system designer to be able to control their values, i.e., to control the period of the sample clock and the duration of the microtick to allow the values for those quantities required by the specification.

9.3.1.1.2 Protocol operation configuration

All configuration data relating to the following protocol operation parameters shall be treated as protocol configuration data:

1. the number of consecutive even/odd cycle pairs with missing clock correction terms *gMaxWithoutClockCorrectionFatal* that will cause the protocol to transition from the *POC:normal active* or *POC:normal passive* state into the *POC:halt* state,
2. the number of consecutive even/odd cycle pairs with missing clock correction terms *gMaxWithoutClockCorrectionPassive* that will cause the protocol to transition from the *POC:normal active* to the *POC:normal passive* state,
3. the number of microticks *pClusterDriftDamping* constituting the cluster drift damping factor used for rate correction within clock synchronization,
4. the number of macroticks *pOffsetCorrectionStart* between the start of the communication cycle and the start of the offset correction within the NIT,

[127] The cycle counter will return to the same value after *gCycleCountMax* + 1 cycles.

5. the number of microticks *pExternOffsetCorrection* constituting the correction term used to correct the calculated offset correction value in the course of external clock synchronization,
6. the number of microticks *pExternRateCorrection* constituting the correction term used to correct the calculated rate correction value in the course of external clock synchronization,
7. the number of microticks *pOffsetCorrectionOut* constituting the upper bound for a permissible offset correction,
8. the number of microticks *pRateCorrectionOut* constituting the upper bound for a permissible rate correction and the maximum drift offset between two nodes operating with non-synchronized clocks for one communication cycle,
9. the Boolean parameter *pAllowHaltDueToClock* that controls the transition to the *POC:halt* state due to a clock synchronization error,
10. the number of consecutive even/odd cycle pairs *pAllowPassiveToActive* during which valid clock synchronization terms must be received before the node transitions from the *POC:normal passive* state to the *POC:normal active* state, including the configuration data to disable transitions from the *POC:normal passive* state to the *POC:normal active* state,
11. the Boolean parameter *pKeySlotOnlyEnabled* that defines whether, after completing startup, a node is restricted to send only in its keyslots or is allowed to transmit in all assigned slots,
12. the Boolean parameter *pKeySlotUsedForStartup* that defines whether the specific node shall send startup frames,
13. the Boolean parameter *pKeySlotUsedForSync* that defines whether the specific node shall send sync frames,
14. the slot ID of the key slot, *pKeySlotID*. A node that does not have a key slot would configure *pKeySlotID* to zero (a value which will never match any actual static slot ID). The effect of such a zero configuration for *pKeySlotID* is that no static slot has the characteristics of the key slot (and thus the node, in effect, does not have a key slot). The key slot, if the node has one, must be assigned to the node by the CHI in all cycles. In particular, the slot ID indicated by a non-zero configuration of *pKeySlotID* is assigned to the node (in the sense that for that slot the variable *vTCHI!Assignment* is set to ASSIGNED) on all configured channels in all communication cycles,
15. the slot ID of the second key slot, *pSecondKeySlotID* in which a second startup frame shall be sent when operating as coldstart node in a TT-L or TT-E cluster,
16. the number of microticks *pDecodingCorrection* used by the node to calculate the primary time reference point,
17. the enumeration *pChannels* that indicates the channels to which the node is connected. The configuration of channels supported by the device, *pChannels*, has a unique characteristic in that it may affect significant hardware details of the implementation. Dual channel devices must be able to support single channel operation and single channel devices must be configurable to work on either channel A or channel B. As a result, it is required that an implementation be able to configure *pChannels*, but the ability to configure this more than once while the CC is in the power on state (see section 2.1.1) is not required. This mechanism does not need to be available in *POC:config*. A device may not, however, allow this parameter to be modified in the *POC:ready*, *POC:normal active*, *POC:normal passive*, or *POC:halt* states, or in any of the states that are defined in the WAKEUP and STARTUP macros of the POC process,
18. the maximum number of consecutive low bits *gdCASRxLowMax* which the node would accept as a valid CAS symbol,
19. the number of bits *gdIgnoreAfterTx* for which bit strobing is paused after a transmission,
20. the number of microticks *pDelayCompensation[A]* used to include the channel A specific reception delay in the calculation of the primary time reference point,
21. the number of microticks *pDelayCompensation[B]* used to include the channel B specific reception delay in the calculation of the primary time reference point,

22. the number of two-byte words *gPayloadLengthStatic* contained in the payload segment of a static frame,
23. the number of bits *gdTSSTransmitter* within the transmission start sequence.
24. the maximum number *gSyncFrameIDCountMax* of distinct sync frame identifiers that may be present in a given cluster,
25. the optional[128] Boolean parameter *pFallBackInternal* that defines whether a time gateway sink node will switch to local clock operation when synchronization with the time gateway source node is lost (*pFallBackInternal* = true) or will instead go to *POC:halt* (*pFallBackInternal* = false),[129]
26. the optional[130] Boolean parameter *pExternalSync* that defines whether the node begins operation[131] with the clock synchronization externally synchronized (*pExternalSync* = true) or not externally synchronized (*pExternalSync* = false),
27. the Boolean parameter *pTwoKeySlotMode* that defines whether the node operates as a coldstart node in a TT-E or TT-L cluster.

9.3.1.1.3 Wakeup and startup configuration

All configuration data relating to the following wakeup and startup parameters shall be treated as protocol configuration data:

1. the number of bits *gdWakeupRxIdle* used by the node to test the duration of the received idle or 'active high' parts of a wakeup,
2. the number of bits *gdWakeupRxLow* used by the node to test the duration of the received 'active low' parts of a wakeup,
3. the number of bits *gdWakeupRxWindow* used by the node to test the duration of a received wakeup,
4. the number of bits *gdWakeupTxIdle* used by the node to transmit the idle part of the wakeup symbol,
5. the number of bits *gdWakeupTxActive* used by the node to transmit the 'active low' part of the wakeup symbol and the 'active low' and 'active high' parts of a WUDOP,
6. the number of wakeup symbols *pWakeupPattern* to be sent by the node to create a wakeup pattern,
7. the enumeration *pWakeupChannel* that indicates on which channel a wakeup symbol shall be sent upon issuing the WAKEUP command,
8. the number of macroticks *pMacroInitialOffset[A]* between a static slot boundary and the subsequent macrotick boundary of the secondary time reference point based on the nominal macrotick duration,
9. the number of macroticks *pMacroInitialOffset[B]* between a static slot boundary and the subsequent macrotick boundary of the secondary time reference point based on the nominal macrotick duration,
10. the number of microticks *pMicroInitialOffset[A]* between the secondary time reference point on channel A and the macrotick boundary immediately following the secondary time reference point,
11. the number of microticks *pMicroInitialOffset[B]* between the secondary time reference point on channel B and the macrotick boundary immediately following the secondary time reference point,
12. the number of microticks *pdAcceptedStartupRange* constituting the expanded range of measured deviation for startup frames during integration,
13. the number of microticks *pdListenTimeout* constituting the upper limit for the startup and wakeup listen timeout,
14. the maximum number of times *gColdstartAttempts* that a node is permitted to attempt to start the cluster by initiating schedule synchronization,

---

[128] The Boolean parameter is only required in implementations that implement a time gateway sink.
[129] This parameter shall not be set to true if the cluster contains more than one TT-E coldstart node.
[130] The Boolean parameter is only required in implementations that implement a time gateway sink.
[131] Note that it is possible that the actual synchronization mode of the cluster (as represented by the variable *vExternalSync*) can change from externally synchronized to not externally synchronized during the operation of the cluster.

15. the upper limit *gListenNoise* of the number of startup and wakeup listen timeouts in the presence of noise.

9.3.1.1.4 Network Management Vector configuration

All configuration data relating to the following Network Management Vector parameters shall be treated as protocol configuration data:

1. the number of bytes *gNetworkManagementVectorLength* contained in the network management vector,
2. the Boolean parameter *pNMVectorEarlyUpdate* that defines the segment after which the accrued network management vector is exported to the CHI.

9.3.1.2 Protocol control data

9.3.1.2.1 Control of the protocol operation control

The CHI shall provide means for the host to send the following protocol control commands to the POC process of the protocol. The conditions under which these commands will be acted upon or ignored are defined in Chapter 9 and Table 2-1:[132]

1. an ALLOW_COLDSTART command that activates the capability of the node to coldstart the cluster,
2. an ALL_SLOTS command that controls the transition from the key slot only mode to the all slots transmission mode,
3. a CONFIG command that causes a transition of the POC process from either the *POC:default config* state or the *POC:ready* state to the *POC:config* state,
4. a CONFIG_COMPLETE command that causes a transition of the POC process from the *POC:config* state to the *POC:ready* state,
5. a FREEZE command that causes a transition of the POC process from any POC state to the *POC:halt* state,
6. an IMMEDIATE_READY command that causes an immediate transition of the POC process to the *POC:ready* state,
7. a RUN command that initiates the startup procedure,
8. a DEFAULT_CONFIG command that causes a transition of the POC process from the *POC:halt* state to the *POC:default config* state,
9. a DEFERRED_HALT command that causes a transition of the POC to the *POC:halt* state,
10. a WAKEUP command that initiates the wakeup procedure,
11. a DEFERRED_READY command that causes a transition of the POC process to the *POC:ready* state from all states except *POC:default config*, *POC:config*, *POC:ready*, *POC:halt*,
12. a CLEAR_DEFERRED command that removes any pending DEFERRED_READY or DEFERRED_HALT command.

9.3.1.2.2 Control of MTS and WUDOP transmission

The CHI shall provide means for the host

1. To control the transmission of MTS symbols on channel A within the symbol window. To perform this the CHI interacts with the protocol engine via the variable *vTransmitMTS_A* (as defined within the MAC_A process),

[132] The CHI does not buffer host commands and issue them to the POC at a later time – they are essentially passed "immediately" to the POC process.

2. To control the transmission of MTS symbols on channel B within the symbol window. To perform this the CHI interacts with the protocol engine via the variable *vTransmitMTS_B* (as defined within the MAC_B process),
3. To control the transmission of WUDOPs on channel A within the symbol window. To perform this the CHI interacts with the protocol engine via the variable *vTransmitWUDOP_A*[133] (as defined within the MAC_A process),
4. To control the transmission of WUDOPs on channel B within the symbol window. To perform this the CHI interacts with the protocol engine via the variable *vTransmitWUDOP_B* (as defined within the MAC_B process).

The control of MTS transmission shall support, at a minimum, the ability of the host to request a single MTS transmission (i.e., a "one-shot" or manual MTS request).

The control of the WUDOP transmission shall support the ability of the host to request a single WUDOP transmission (i.e., a "one-shot" or manual WUDOP request), but shall also support an automatic request by which the host can identify a set of FlexRay communication cycles in which the CC will transmit a WUDOP in the symbol window without additional host interaction. This automatic mechanism shall operate in addition to the manual mechanism (i.e., a WUDOP should be sent if either the manual or automatic mechanisms, or both, indicate that a WUDOP should be sent).

The configuration of the specific cycles in which the automatic transmission of WUDOPs takes place shall support, at a minimum, the ability to define sets of communication cycles which equal the sets which can be defined using a Cycle_Repetition and a Cycle_Offset to determine the configuration:

Automatically transmit a WUDOP in the symbol window if

vCycleCounter mod Cycle_Repetition = Cycle_Offset

with

Cycle_Repetition selected from the set of {1, 2, 4, 5, 8, 10, 16, 20, 32, 40, 50, 64}[134]

Cycle_Offset selected from the set {0 … 63} with Cycle_Offset < Cycle_Repetition

It is not required that an implementation supports independent sets of repetition and offset parameters for each channel (i.e., a single set of parameters controlling automatic WUDOP generation for both channels is acceptable). It is acceptable, but not required, that the repetition and offset parameters controlling automatic WUDOP generation be modifiable during operation (i.e., when the node is in the *POC:normal active* state). It is required, however, that automatic WUDOP generation be independently controllable (i.e., switched on or off) on a per channel basis when the node is in the *POC:normal active* state.[135]

A transition into the *POC:ready* state shall cause the CHI to reset any pending one-shot transmissions of WUDOP's or MTS's.[136]

Transmissions in the symbol window must be coordinated at a system level, i.e., the system designer must ensure for each channel that in any given cycle at most one node will transmit either an MTS or a WUDOP in the symbol window on that channel.

9.3.1.2.3 Control of external clock synchronization

The CHI shall provide means for the host

[133] This variable represents the control value of both the automatic and manual mechanisms for WUDOP transmission.
[134] Note that these particular values of Cycle_Repetition are those that could be achieved by combining a "power of 2" filter with a "count to 5" filter.
[135] It is also allowable that automatic WUDOP generation can be switched on or off in other POC states, but the actual generation of WUDOPs will only take place when the node is in the *POC:normal active* state.
[136] This prevents unintentional WUDOP or MTS transmissions left over from previous operation for TT-E coldstart nodes (for example, if an IMMEDIATE_READY command is followed by a RUN command).

1. to control the application of the external offset correction parameter *pExternOffsetCorrection* using the control value *vExternOffsetControl*,
2. to control the application of the external rate correction parameter *pExternRateCorrection* using the control value *vExternRateControl*.

A transition into the *POC:ready* state shall cause the CHI to set the variables that control the operation of the external rate and offset correction (i.e., *vExternRateControl* and *vExternOffsetControl*) such that no external rate or offset correction is requested.

9.3.1.3 Protocol status data

The CHI shall provide the protocol status information described in the following subsections.

The following subsections define a number of indicators. These indicators are a type of status information that can take on two distinct values, labeled as "set" and "reset". An indicator is set (i.e., given the "set" value) by the CHI when certain events occur in the execution of the protocol, and are, in general, reset (i.e., given the "reset" value) by the host.

An implementation shall ensure that each indicator described in the following subsections is reset under at least one of the following conditions:

- every transition into the *POC:default config* state
- every transition into the *POC:config* state
- every transition out of the *POC:config* state.

The specific behavior of an implementation with respect to the reset behavior of each indication is implementation dependent.

9.3.1.3.1 Protocol operation control status

The following protocol operation control status variables shall be provided in the CHI:

1. the status variable *vPOC!State* (as maintained by the POC process),
2. the flag *vPOC!Freeze* (as maintained by the POC process),
3. the flag *vPOC!CHIReadyRequest* (as maintained by the POC process),
4. the flag *vPOC!CHIHaltRequest* (as maintained by the POC process),
5. the flag *vPOC!ColdstartNoise* (as maintained by the POC process),
6. the status variable *vPOC!SlotMode* (as maintained by the POC process),
7. the status variable *vPOC!ErrorMode* (as maintained by the POC process),
8. the number of consecutive even/odd cycle pairs *vAllowPassiveToActive* that have passed with valid rate and offset correction terms, but with the node still in *POC:normal passive* state due to a host configured delay to *POC:normal active* state (as maintained by the POC process),
9. the status variable *vPOC!WakeupStatus* (as maintained by the POC process),
10. the status variable *vPOC!StartupState* (as maintained by the POC process),
11. the optional[137] flag *vExternalSync* (as maintained by the CSP process).

9.3.1.3.2 Wakeup and startup status

The following startup status variable shall be provided in the CHI:

1. the number of remaining coldstart attempts *vRemainingColdstartAttempts* (as maintained by the POC process).

The CHI shall provide indicators for the following wakeup and startup events:

[137] The status variable is only required in implementations that implement a time gateway sink.

1. a coldstart abort indicator that shall be set upon 'set coldstart abort indicator in CHI' (in accordance with the POC process) and reset under control of the host,
2. a wakeup pattern received indicator for channel A that shall be set upon 'set wakeup received indicator on A in CHI' (in accordance with the FSP_A process) and reset under control of the host,
3. a wakeup pattern received indicator for channel B that shall be set upon 'set wakeup received indicator on B in CHI' (in accordance with the FSP_B process) and reset under control of the host.

9.3.1.3.3 Communication cycle timing status

The following communication cycle timing variables shall be provided in the CHI:

1. the macrotick *vMacrotick* (as maintained by the MTG process),
2. the cycle counter *vCycleCounter* (as maintained by the MTG process),
3. the slot counter *vSlotCounter* for channel A (as maintained by the MAC_A process),
4. the slot counter *vSlotCounter* for channel B (as maintained by the MAC_B process).

The values provided to the host by the CHI for *vMacrotick*, *vCycleCounter* and *vSlotCounter* for channel A and B shall be valid during the states *POC:normal active* and *POC:normal passive*.

A snapshot of the following communication cycle timing variables shall be provided in the CHI:

1. the rate correction value *vInterimRateCorrection* (in accordance with the CSP process).
   - If *vExternalSync*[138] is false the rate correction value is based on the internally calculated values, is not limited by *pRateCorrectionOut*, and does not include any external rate correction value.
   - If *vExternalSync* is true the rate correction value is imported from the time gateway source and includes the limitation and external rate correction values as configured in the time gateway source.
2. the offset correction value *vInterimOffsetCorrection* (as maintained by the CSP process).
   - If *vExternalSync* is false the offset correction value is based on the internally calculated values, is not limited with *pOffsetCorrectionOut*, and does not include any external offset correction value.
   - If *vExternalSync* is true the offset correction value is imported from the time gateway source and includes the limitation and external offset correction values as configured in the time gateway source.

The CHI shall provide indicators for the following communication cycle timing events:

1. a sync frame overflow indicator that shall be set upon 'set sync frame overflow indicator in CHI' (in accordance with the CSP process) and reset under control of the host,
2. a *pLatestTx* violation status indicator for channel A that shall be set upon 'set *pLatestTx* violation status indicator on A in CHI' (in accordance with the MAC_A process), and reset under control of the host,
3. a *pLatestTx* violation status indicator for channel B that shall be set upon 'set *pLatestTx* violation status indicator on B in CHI' (in accordance with the MAC_B process), and reset under control of the host,
4. a transmission across boundary violation status indicator for channel A that shall be set upon 'set transmission across slot boundary violation indicator on A in CHI' (in accordance with the FSP_A process),
5. a transmission across boundary violation status indicator for channel B that shall be set upon 'set transmission across slot boundary violation indicator on B in CHI' (in accordance with the FSP_B process).

[138] If the optional flag *vExternalSync* does not exist the communication controller shall behave as if *vExternalSync* is equal to false. Refer to section 1.10.3.

9.3.1.3.4 Synchronization frame status

A snapshot of the following information, derived from the *vsSyncIDListA* and *vsSyncIDListB* variables provided by the CSP process, shall be provided in the CHI:

1. A list containing the IDs of the sync frames received or transmitted on channel A within the even communication cycle as well as the number of valid entries contained in this list,
2. A list containing the IDs of the sync frames received or transmitted on channel B within the even communication cycle as well as the number of valid entries contained in this list,
3. A list containing the IDs of the sync frames received or transmitted on channel A within the odd communication cycle as well as the number of valid entries contained in this list,
4. A list containing the IDs of the sync frames received or transmitted on channel B within the odd communication cycle as well as the number of valid entries contained in this list.

The information shall be updated no sooner than the start of the NIT and no later than 10 macroticks after the start of the offset correction phase of the NIT. Note that this implies that for some NIT configurations the data update may complete after the start of the next cycle.

The following synchronization frame variable shall be provided in the CHI:

1. the number of consecutive even/odd cycle pairs *vClockCorrectionFailed* that have passed without clock synchronization having performed an offset or a rate correction due to lack of synchronization frames (as maintained by the POC process).

9.3.1.3.5 Startup frame status

A snapshot of the following information, derived from the *vStartupPairs* variable provided by the CSP process, shall be provided in the CHI:

1. the number of channel aligned startup frame pairs received or transmitted during the previous double cycle, aggregated across both channels.

The availability of this information depends on the synchronization method and role of the node in the cluster. For TT-E coldstart nodes the information shall be updated no sooner than one microtick before the end of the NIT in the odd cycle and no later than 10 macroticks after the start of the next cycle. For all other nodes the information shall be updated no sooner than the start of the NIT in the odd cycle and no later than 10 macroticks after the start of the offset correction phase of the NIT. Note that this implies that for some NIT configurations the data update may complete after the start of the next cycle.

9.3.1.3.6 Symbol window status

A snapshot of the following symbol window variables shall be provided in the CHI for the slot status *vSS* established for each channel at the end of the symbol window:

1. the flag *vSS!ValidMTS* for channel A (in accordance with the FSP_A process),
2. the flag *vSS!ValidMTS* for channel B (in accordance with the FSP_B process),
3. the flag *vSS!SyntaxError* for channel A (in accordance with the FSP_A process),
4. the flag *vSS!SyntaxError* for channel B (in accordance with the FSP_B process),
5. the flag *vSS!BViolation* for channel A (in accordance with the FSP_A process),
6. the flag *vSS!BViolation* for channel B (in accordance with the FSP_B process),
7. the flag *vSS!TxConflict* for channel A (in accordance with the FSP_A process),
8. the flag *vSS!TxConflict* for channel B (in accordance with the FSP_B process).

The following list indicates the behavior of the CHI depending on the activities detected on the RxD input:

- The low phases of a WUDOP might cause an indication of a valid MTS even if only a WUDOP was actually present. This may or may not happen (depending on the data rate, which affects the duration of what is accepted as a valid MTS).

- Because the low phases of a WUDOP can be considered as a valid MTS, the reception of multiple valid MTS's within a symbol window is not indicated as a syntax error.
- Although the WUDOP is transmitted in the symbol window, an indication that a wakeup has been received may not occur until sometime during the NIT (i.e., the symbol window status may become available to the host before the indication that a wakeup has been received becomes available to the host).
- In the normal situation, the reception of an MTS or WUDOP will not result in boundary violations or syntax errors (i.e., those conditions represent exceptional circumstances such as additional communication elements, noise on the link, etc.).
- It is possible to detect a valid MTS, or to detect a wakeup, even in the presence of syntax errors or boundary violations.

9.3.1.3.7 NIT status

A snapshot of the following NIT variables shall be provided in the CHI for the slot status *vSS* established for each channel at the end of the NIT:

1. the flag *vSS!SyntaxError* for channel A (in accordance with the FSP_A process),
2. the flag *vSS!SyntaxError* for channel B (in accordance with the FSP_B process),
3. the flag *vSS!BViolation* for channel A (in accordance with the FSP_A process),
4. the flag *vSS!BViolation* for channel B (in accordance with the FSP_B process).

9.3.1.3.8 Aggregated channel status

The aggregated channel status provides the host with an accrued status of channel activity for all communication slots regardless of whether they are assigned for transmission or subscribed for reception. The status is aggregated over a period that is freely definable by the host.

The CHI shall provide indicators for the following channel activity events:

1. a channel specific valid frame indicator for channel A that shall be set if a valid frame was received in any static or dynamic slot on channel A (i.e., one or more static or dynamic slots had *vSS!ValidFrame* equal to true) and reset under control of the host,
2. a channel specific valid frame indicator for channel B that shall be set if a valid frame was received in any static or dynamic slot on channel B (i.e., one or more static or dynamic slots had *vSS!ValidFrame* equal to true) and reset under control of the host,
3. a channel specific syntax error indicator for channel A that shall be set if one or more syntax errors were observed on channel A (i.e., one or more static or dynamic slots including symbol window and NIT had *vSS!SyntaxError* equal to true) and reset under control of the host,
4. a channel specific syntax error indicator for channel B that shall be set if one or more syntax errors were observed on channel B (i.e., one or more static or dynamic slots including symbol window and NIT had *vSS!SyntaxError* equal to true) and reset under control of the host,
5. a channel specific content error indicator for channel A that shall be set if one or more frames with a content error were received on channel A in any static or dynamic slot (i.e., one or more static or dynamic slots had *vSS!ContentError* equal to true) and reset under control of the host,
6. a channel specific content error indicator for channel B that shall be set if one or more frames with a content error were received on channel B in any static or dynamic slot (i.e., one or more static or dynamic slots had *vSS!ContentError* equal to true) and reset under control of the host,
7. a channel specific additional communication indicator for channel A that shall be set if one or more valid frames were received on channel A in slots that also contained any additional communication during the observation period (i.e., one or more slots had *vSS!ValidFrame* equal to true and any combination of either *vSS!SyntaxError* equal to true or *vSS!ContentError* equal to true or *vSS!BViolation* equal to true) and reset under control of the host,

8. a channel specific additional communication indicator for channel B that shall be set if one or more valid frames were received on channel B in slots that also contained any additional communication during the observation period (i.e., one or more slots had *vSS!ValidFrame* equal to true and any combination of either *vSS!SyntaxError* equal to true or *vSS!ContentError* equal to true or *vSS!BViolation* equal to true) and reset under control of the host,
9. a channel specific slot boundary violation indicator for channel A that shall be set if one or more slot boundary violations were observed on channel A (i.e., one or more static or dynamic slots including symbol window and NIT had *vSS!BViolation* equal to true) and reset under control of the host,
10. a channel specific slot boundary violation indicator for channel B that shall be set if one or more slot boundary violations were observed on channel B (i.e., one or more static or dynamic slots including symbol window and NIT had *vSS!BViolation* equal to true) and reset under control of the host,
11. a channel specific transmission conflict indicator for channel A that shall be set if one or more transmission conflicts were observed on channel A (i.e., one or more static or dynamic slots and/or the symbol window had *vSS!TxConflict* equal to true) and reset under control of the host,
12. a channel specific transmission conflict indicator for channel B that shall be set if one or more transmission conflicts were observed on channel B (i.e., one or more static or dynamic slots and/or the symbol window had *vSS!TxConflict* equal to true) and reset under control of the host.

9.3.1.3.9 Dynamic segment status

The following dynamic segment status variables shall be provided in the CHI:

1. a channel specific value which represents the slot counter of the last frame transmitted by the node on channel A in the dynamic segment as reflected by the variable *vLastDynTxSlot* in the MAC_A process. It is updated at the end of the dynamic segment and would have a value of zero if no frame was transmitted during the dynamic segment.[139]
2. a channel specific value which represents the slot counter of the last frame transmitted by the node on channel B in the dynamic segment as reflected by the variable *vLastDynTxSlot* in the MAC_B process. It is updated at the end of the dynamic segment and would have a value of zero if no frame was transmitted during the dynamic segment.
3. a channel specific dynamic resynchronization attempted flag as reflected by the variable *vDynResyncAttempt* in the MAC_A process which is set if either a slot was skipped, or a transmission might have been blocked (i.e., the *zNoTxSlot* variable was set to true, whether or not this actually resulted in one less transmission in the system), otherwise it is reset. It is updated at the end of the dynamic segment.
4. a channel specific dynamic resynchronization attempted flag as reflected by the variable *vDynResyncAttempt* in the MAC_B process which is set if either a slot was skipped, or a transmission might have been blocked (i.e., the *zNoTxSlot* variable was set to true, whether or not this actually resulted in one less transmission in the system), otherwise it is reset. It is updated at the end of the dynamic segment.

9.3.2 Message data interface

The message data interface addresses the

- management of the communication slots in which the node shall transmit messages,
- subscription of messages the host wants to receive,
- exchange of message data between the host and the protocol engine within the node.

Message transmission pertains to the message data flow from the host out to a FlexRay network, and message reception pertains to the message data flow from a FlexRay network in to the host.

[139] This variable can be used to determine if the frame corresponding to a given slot in the dynamic segment was actually transmitted. This is especially beneficial for continuous transmission mode since then the "frame transmitted" flag is less useful.
In the event of an aborted transmission due to a *pLatestTx* violation in the dynamic segment, this value will indicate the ID of the frame that was aborted.

9.3.2.1 Communication slot assignment

A node in the *POC:normal active* state is able to transmit messages by sending frames on one or both channels of the FlexRay bus and is able to receive messages by receiving frames on one or both channels of the FlexRay bus.

The information on when a node shall send or receive a message is called communication slot assignment.

Communication slot assignment shall be done for each available channel.

A specific communication slot of a specific communication cycle can be identified by the pair of a slot number and a communication cycle number.

Slot multiplexing, i.e. assigning slots having the same slot identifier but different communication cycle numbers to different nodes, is allowed by the protocol in the static segment for slots which are not configured as key slots. The configuration of assignment must ensure that transmission in the cluster is conflict-free.[140]

Slot multiplexing is allowed for all slots in the dynamic segment. It is up to the application or the configuration to ensure that transmission in the cluster is conflict-free.

9.3.2.2 Communication slot assignment for transmission

A specific TDMA slot (communication slot) in either the static or dynamic segment of a specific communication cycle shall be assigned to a node for transmission by assigning the corresponding slot identifier and communication cycle number for a channel to the node according to the constraints listed in sections 5.1.3.1 and 5.1.4.1.

A node in the *POC:normal active* state will always transmit a null frame or a non-null (data) frame in a slot assigned to this node for transmission within the static segment. A node in the *POC:normal active* state will always transmit a non-null (data) frame in a slot assigned to this node for transmission within the dynamic segment if an active transmit message buffer is found (see section 9.3.2.6.5).

9.3.2.2.1 Cycle-independent and cycle-dependent slot assignment

The CHI shall provide the possibility to assign communication slots of a channel to the node for transmission independent of the cycle number, i.e. all slots sharing the slot ID in all communication cycles are assigned to this node on this channel. This method of assignment is referred to as "cycle-independent slot assignment".

The CHI shall provide the possibility to assign individual slots (identified by the pair of a slot number and a cycle counter number) or sets of slots (identified by a slot number and a set of communication cycle numbers) of a channel to the node for transmission. This method of assignment is referred to as "cycle-dependent slot assignment". The CHI shall provide a mechanism to enable or disable the possibility of cycle-dependent slot assignment for slots located in the static segment.[141]

The enable / disable mechanism for cycle-dependent slot assignment shall be available to the host only while the node is in the *POC:config* state.

If cycle-dependent slot assignment is disabled, the CHI shall only allow cycle-independent slot assignment for communication slots in the static segment. In this case the node will transmit a frame in all slots of the static segment sharing the slot number of the assigned slot in all communication cycles.

---

[140] It is also possible to achieve slot multiplexing in the static segment by reconfiguring assignment during the operation in the system. In this case it is up to the application to ensure conflict-free transmission.

[141] The enable / disable mechanism allows the user to select between static segment behavior consistent with the previous versions of the protocol (where a node that transmits in a slot with a specific slot number in any cycle must transmit either a non-null frame or a null frame in all slots with this number in all cycles) and new static segment behavior that allows a node to transmit in a slot with a specific slot number in only some cycles (i.e., it is no longer necessary that a node that transmits in a slot in a particular cycle must transmit in all slots with this number).

In the static segment

when a slot in a communication cycle occurs and this slot is assigned to a node, the node shall transmit either a non-null frame or a null frame in that slot. Specifically, a null frame will be sent if there is no data ready, or if there is no transmit buffer configured for this slot (see section 9.3.2.7.1).

In the dynamic segment

when a slot in a communication cycle occurs and this slot is assigned to a node, the node only transmits a non-null frame in this slot if an active transmit message buffer is found (see section 9.3.2.6.5).

In both segments

The system designer must ensure that no two nodes transmit in the same slot.

9.3.2.2.2 Transmission slot assignment list

The set of all slots assigned to a node for transmission is called the "transmission slot assignment list".

An implementation may explicitly implement the transmission slot assignment list or derive it dynamically from the transmit buffer assignment (see section 9.3.2.7.1) or implement a mixture of explicit and implicit assignment.

In any case the implementation must ensure that all requirements for the transmission slot assignment list in this chapter are fulfilled. This is especially true for the supported sets of slots which can be assigned to the node for transmission. The transmission slot assignment list has to support all combinations of sets of slots for which transmit buffers can be configured.

The CHI has to ensure that at the beginning of each slot it provides up to date information to the protocol engine as described in section 9.3.2.7.2.

In the static segment this is especially true for the header CRC. If there is no transmit buffer configured for a specific entry of the transmission slot assignment list, the corresponding header CRC for this specific transmission slot assignment list entry must be part of the transmission slot assignment list so that a valid null frame can be sent. Further descriptions of the mechanisms in this chapter assume that the transmission slot assignment list includes the corresponding header CRC for each entry.

An implementation shall provide the ability to modify the transmission slot assignment list in all states other than *POC:default config*, and shall prevent modification of the transmission slot assignment list in the *POC:default config* state.

An implementation shall provide a configurable mechanism to prevent the host from modifying the transmission slot assignment list in states other than *POC:config*. Configuration (i.e., enabling or disabling) of this mechanism shall only be possible in the *POC:config* state. It is allowed, but not required, that this configuration mechanism be the same mechanism that allows configuration of write access to the message buffer configuration information described in section 9.3.2.5.1.

Modification of the transmission slot assignment list shall only be possible in states other than *POC:config* if this capability was explicitly enabled during the *POC:config* state.

9.3.2.2.3 Key slot assignment

For key slots only cycle-independent slot assignment is allowed. Key slots must be assigned to a node for all connected channels, i.e. all slots of all connected channels sharing the key slot IDs in all communication cycles are assigned to this node.

The modification of the assignment of key slots shall only be possible during the *POC:config* state. The node shall ignore attempts to change the assignment of a key slot in all other states.

9.3.2.3 Communication slot assignment for reception

A node in the *POC:normal active* state can receive all frames and therefore all messages on the FlexRay bus. Frames which are relevant for the operation of the protocol (sync frames, for example) will be processed by the protocol engine automatically without the need of explicit assignment of the corresponding communication slots for reception.

This specification gives no guidance on how communication slot assignment for reception shall be implemented. Section 9.3.2.7.1, however, descibes requirements for message buffers used for reception.

9.3.2.4 Conflicting communication slot assignment for reception and transmission

In the static segment if a node has been configured to receive a message in a specific slot it shall only do so if this slot has not also been assigned to the node for transmission. Transmission of messages in the static segment always has precedence over reception.

In the dynamic segment transmission has precedence over reception in a specific slot if there is at least one transmit message buffer with valid payload data configured for this specific slot.

9.3.2.5 Non-queued message buffers

Message transmission and reception operate on message buffers. A non-queued message buffer is a structure which consists of:

- message buffer configuration data,
- message buffer status data, and
- message data.

Upon a transition from either the *POC:normal active* or *POC:normal passive* state to either the *POC:halt* or *POC:ready* state the CHI shall continue to provide the host access to the data that it would have provided had the POC remained in the *POC:normal active* or *POC:normal passive* states. Note that at this point the CHI will no longer update the data as the protocol engine will stop providing new data.

The behavior of the CHI upon attempted host access to buffer status or payload data that has never been updated by the CHI because no data was provided by the protocol engine is implementation dependent. It is required that the access to such data does not give the appearance that data was received or transmitted when such a reception or transmission did not actually take place.

9.3.2.5.1 Message buffer configuration data

The message buffer configuration data shall contain at least the following parameters for each message buffer:

- a type indication (transmit or receive message buffer),
- the channel identifier (A, B, or, for dual channel devices, A&B) on which the node shall transmit or receive a message using this buffer,
- the slot identifier of the communication slot in which the node shall transmit or receive a message using this buffer,
- the set of communication cycles in which the node shall transmit or receive a message using data provided in this message buffer.

Additionally a message buffer configured for transmission needs to contain the following configuration parameters

- the length of the available message data in the buffer (*MessageLength*),[142]
- the payload preamble indicator (for the network management service or message ID filtering),
- the header CRC,

- a transmission mode indicator (single shot mode or continuous mode).

The previously identified buffer configuration information is divided into two classes:

Class 1 Buffer Configuration Data:

- the type indication
- the channel identifier
- the slot identifier
- the set of communication cycles

Class 2 Buffer Configuration Data:

- the length of the available message data
- the payload preamble indicator
- the header CRC
- the transmission mode indicator

The host shall have read access to both classes of message buffer configuration data independently of the protocol state.

The host shall have write access to both classes of message buffer configuration data in the *POC:config* state.

An implementation shall allow configuration of message buffers in all states other than *POC:default config*, and shall prevent configuration of message buffers in the *POC:default config* state. An implementation shall provide a configurable mechanism to prevent the host from writing certain classes of message buffer configuration data in states other than *POC:config*. The configuration of this mechanism, i.e., the selection of which classes of message buffer configuration data can be written, shall only be possible in the *POC:config* state. At a minimum, an implementation must support the following configurations:

- No write access to either Class 1 or Class 2 message buffer configuration data while in states other than *POC:config* and *POC:default config*
- Write access to both Class 1 and Class 2 message buffer configuration data while in states other than *POC:config* and *POC:default config*
- Write access to Class 2 message buffer configuration data but no write access to Class 1 buffer configuration data while in states other than *POC:config* and *POC:default config*

An implementation must be capable of applying the previous configurations to all non-queued message buffers[143], but an implementation may provide more fine-grained control, allowing sets of message buffers to have different restrictions.

The implementation must ensure that it is not possible for the host to change the configuration data of a message buffer while the buffer is being used according to section 9.3.2.7.2 or while it is updated by the controller.

9.3.2.5.2 Message buffer status data

The message buffer status shall be able to provide the following information to the host

- slot status,

---

[142] In the static segment *MessageLength* describes the number of two-byte words that are provided to the protocol engine by the CHI. The macro ASSEMBLE_STATIC_FRAME assures that a frame with the correct length (i.e., *gPayloadLengthStatic*) is transmitted by the protocol engine.
In the dynamic segment this value describes the actual number of two-byte words in the frame to be transmitted.

[143] An exception to this requirement applying to all buffers is that it is allowable that message buffers exclusively associated with key slots are always restricted from configuration in states other than *POC:config*.

- a slot status updated indicator and
- channel indication of the channel which the slot status refers to.

The message buffer status shall be able to indicate the status information listed in section 9.3.2.7.3 for a transmit message buffer or the status information listed in section 9.3.2.8.2.1 for a receive message buffer. The CHI shall indicate that the slot status information of a message buffer has been updated by setting the corresponding slot status updated indicator to true.

When the host (re)configures a message buffer (i.e., when the host performs a write access to the class 1 or class 2 configuration data for a buffer), the CHI shall set the slot status updated indicator of the corresponding message buffer to false.

The CHI shall set the slot status updated indicator to false for all message buffers upon a transition from *POC:ready* state to either *POC:coldstart listen*, *POC:external startup* or *POC:integration listen* (i.e., upon a transition of the variable *vPOC!State* from READY to STARTUP).

9.3.2.5.3 Message buffer payload data and payload data valid flag

The payload data to be transmitted in a frame or the payload data which a node receives will be stored in the message buffer data section of the corresponding message buffer (see 9.3.2.8.1 and 9.3.2.7.1).

The CHI will grant the following access rights to the message buffer payload data:

- For transmit buffers the host shall have read and write access to the data. For receive buffers the host shall have read access to the data. The CHI will write the data on reception of the corresponding frame when the data is provided by the protocol engine.

The CHI must ensure that it always provides a consistent set of configuration and status and message data to the host in case of receive message buffers and to the protocol engine in case of transmit message buffers.

Each message buffer shall provide a Boolean payload data valid flag associated with the message buffer payload data.

The host shall be granted the same access rights to this flag as for the message buffer payload data.

By setting this flag to true in a transmit message buffer the host indicates that the message buffer payload data in the message buffer is ready for transmission, i.e. the payload data is "valid". The CHI can change the value of the payload data valid flag when the corresponding frame has been sent depending on the transmission mode indicator (see section 9.3.2.7.1).

For receive buffers the CHI sets the payload data valid flag to true when the message buffer payload data actually contains the payload data of a received frame.

When the host (re)configures a message buffer (i.e., when the host performs a write access to the class 1 or class 2 configuration data for a buffer), the CHI shall set the payload data valid flag of the corresponding message buffer to false.

The CHI shall set the payload data valid flag to false for all receive message buffers upon a transition from *POC:ready* state to either *POC:coldstart listen*, *POC:external startup* or *POC:integration listen* (i.e., upon a transition of the variable *vPOC!State* from READY to STARTUP). The payload data valid flag for transmit message buffers shall not be modified upon a transition from *POC:ready* state to any state.[144] [145]

The CHI shall set the payload data valid flag to false for all transmit message buffers upon a transition into the *POC:ready* state from any state other than *POC:config*, *POC:wakeup listen*, *POC:wakeup send*, and

[144] This allows an application to pre-configure payload data for transmission in the *POC:ready* or *POC:config* states, i.e. to prepare buffers for transmission before issuing the RUN command.

[145] The system designer should be aware that after issuing a RUN command the payload data valid flag is only cleared for receive buffers - the payload data valid flag for transmit buffers remains in the original state.

POC:wakeup detect (i.e., when the *vPOC!State* variable changes from either STARTUP, NORMAL_ACTIVE, or NORMAL_PASSIVE to READY).[146]

9.3.2.5.4 Buffer Enabling and Buffer Locking

For the purposes of the selection of buffers, this specification defines two additional concepts related to the availability of buffers - an enabled / disabled status and a locked / unlocked (buffer locking) status.

In this context, a buffer being "enabled" refers to whether or not a buffer's configuration (as described in section 9.3.2.5.1) is complete. A buffer that has not been configured, or is in the process of being configured or reconfigured, is considered disabled. A buffer whose configuration is complete, and is not in the process of being reconfigured, is considered enabled. Note that the enabled / disabled status of a buffer has nothing to do with the host reading or making modifications to the status or payload data of the buffer (see buffer locking, described below).

Buffer locking refers to an optional, implementation-specific mechanism often used to ensure that the host's accesses to a buffer's slot status and payload data information are atomic. When a buffer is locked the CHI will not update the buffer's slot status or payload data information, and will inform the protocol engine (via the *vTCHI!TxMessageAvailable* flag) that no data is available from the buffer. Buffer locking is an optional capability of an implementation, but if present, the locked / unlocked status shall be considered when deciding if a buffer is a candidate (see sections 9.3.2.6.1 and 9.3.2.6.2).

9.3.2.6 Non-queued message buffer identification

In case that the host has configured several non-queued message buffers for transmission and/or reception in a specific slot the CHI has to select an "active message buffer" at the beginning of this specific slot in a deterministic way according to the following general steps which will be detailed in the following subsections:

1. Identify the set of "candidate transmit buffers" and the set of "candidate receive buffers", i.e. message buffers which have been configured for the specific slot where the transmission or reception shall happen.
2. Identify one buffer out of the set of the candidate transmit buffers for the specific channel as the "selected transmit buffer" and identify it as the "active message buffer" for the channel. If no transmit buffer is found then identify one buffer out of the set of the candidate receive buffers for the specific channel as the "selected receive buffer" and identify it as the "active message buffer" for the channel.

These steps must be performed on a per channel basis, i.e., if the communication controller is configured for dual channel operation the identification process must be performed for each channel without considering the result of the other channel. As a result the CHI in this case may identify two active message buffers, one for each channel.[147]

9.3.2.6.1 Candidate transmit message buffer identification

If a slot is assigned to the node according to the transmission slot assignment list the CHI shall evaluate

- the message buffer configuration (type indication, slot identifier, channel identifier and set of communication cycles) and the payload data valid flag of all non-queued message buffers AND
- the current status of the buffer locking (if applicable) AND
- the segment in which the slot with the specific slot identifier is located AND
- the current protocol state *vPOC!State* of the protocol engine AND
- the current slot mode *vPOC!SlotMode* of the protocol engine

to identify the set of the candidate transmit message buffers for this slot.

[146] This prevents unintentional transmissions due to payload data valid flags left over from previous operation (for example, if an IMMEDIATE_READY command is followed by a RUN command).

[147] Note that a message buffer that is configured for both channels can be the only active message buffer.

The following table determines for a given buffer whether it becomes a member of the set of candidate transmit buffers (value true in the "candidate" column). The table uses the term "buffer match" to indicate that

- the type indication of the buffer identifies it as transmit buffer AND
- the slot identifier matches the slot number of the specific slot AND
- the channel identifier is included in *pChannels* AND
- the specific communication cycle is in the set of communication cycles AND
- the message buffer is enabled[148] by the host AND
- if the implementation supports buffer locking[149], the message buffer is not locked by the host.

Buffer match evaluates to true if all the conditions above are met. It evaluates to false if one or more of these conditions are not met.

Buffer match	Segment	vPOC!State	payload data valid flag	vPOC!Slot-Mode	Candidate
true	dynamic	NORMAL_ACTIVE	true	KEYSLOT or ALL_PENDING	false
true	dynamic	NORMAL_ACTIVE	true	ALL	true
true	dynamic	NORMAL_ACTIVE	false	don't care	false
true	dynamic	all but NORMAL_ACTIVE	don't care	don't care	false
true	static	don't care	don't care	don't care	true
false	don't care	don't care	don't care	don't care	false

Table 9-1: Transmit message buffer candidate.

If a slot is not assigned to the node for transmission according to the transmission slot assignment list, then the set of the candidate transmit message buffers for this slot is empty.

9.3.2.6.2 Candidate receive message buffer identification

A non-queued message buffer will be included in the set of candidate receive message buffers for a slot when

- the type indication of the buffer is set to receive buffer AND
- the slot identifier matches the slot number of the specific slot AND
- the channel identifier is included in *pChannels* AND
- the specific communication cycle is in the set of communication cycles AND
- the message buffer is enabled[150] by the host AND
- if the implementation supports buffer locking[151], the message buffer is not locked by the host.

For the static segment only the following additional criterion also applies:

- The CHI does not consider the slot / channel combination to be assigned for transmission (i.e., the value of *vTCHI!Assignment* that is passed to the protocol engine is UNASSIGNED).

If none of the configured receive message buffers fulfills all these conditions, then the set of candidate receive message buffers is empty for the slot.

[148] Please refer to section 9.3.2.5.4.
[149] Please refer to section 9.3.2.5.4.
[150] Please refer to section 9.3.2.5.4.
[151] Please refer to section 9.3.2.5.4.

9.3.2.6.3 Selected transmit buffer identification

In case that more than one message buffer are members of the set of candidate transmit message buffers, the CHI has to select one specific message buffer out of this set. This selection process is implementation dependent, but has to meet all of the following requirements:

- The selection process has to be deterministic so that the host can predict the result.
- If there is at least one candidate transmit message buffer there must be a selected transmit message buffer.
- The selected transmit message buffer must be one of the candidate transmit message buffers.
- If one or more of the transmit candidate message buffers have the payload data valid flag set to true the selected message buffer must be one of these.
- If the set of candidate transmit message buffers is empty, no buffer will become the selected transmit buffer.

9.3.2.6.4 Selected receive buffer identification

In case that more than one message buffer are members of the set of candidate receive message buffers, the CHI has to select one specific message buffer out of this set. This selection process is implementation dependent, but has to meet all of the following requirements:

- The selection process has to be deterministic so that the host can predict the result.
- If there is at least one candidate receive message buffer there must be a selected receive message buffer.
- The selected receive message buffer must be one of the candidate receive message buffers.
- If the set of candidate receive message buffers is empty, no buffer will become the selected receive buffer.

9.3.2.6.5 Active message buffer identification

The active message buffer is determined according to the following rules:

- If there is a selected transmit message buffer then the active message buffer is the selected transmit message buffer.
- If there is no selected transmit message buffer but there is a selected receive message buffer the active message buffer is the selected receive message buffer.
- If there is neither a selected transmit message buffer nor a selected receive message buffer then there is no active message buffer.

Depending on the value of the type indication the active message buffer will be called "active transmit message buffer" or "active receive message buffer" in the following sections.

For a given channel there can be at most one active message buffer at a time - independent of its type.

9.3.2.7 Message transmission

Message transmission is primarily determined by the concept of the transmission slot assignment list (see section 9.3.2.2). Transmit message buffers provide the information (e.g. payload data) which shall be transmitted but in principle transmission could take place without a transmit buffer being configured.

9.3.2.7.1 Transmit buffer configuration

A transmit buffer shall be configured for a transmission based on the channel identifier and the slot assignment information describing in which slot the transmission shall occur.

For frames transmitted in the static segment, the following channel configuration shall be supported:

1. configured for channel A

2. configured for channel B
3. for dual channel devices, configured for channel A and for channel B.

For frames transmitted in the dynamic segment, the following channel configurations shall be supported:

1. configured for channel A
2. configured for channel B.

It shall be possible to configure a transmit buffer for a single slot or for a set of slots sharing the same slot identifier in a configurable set of communication cycles.

The configuration of the set of communication cycles shall support, at a minimum, the ability to define sets of communication cycles which equal the sets which can be defined using a Cycle_Repetition and a Cycle_Offset to determine the configuration:

The transmit buffer is configured for a transmission slot if

vCycleCounter mod Cycle_Repetition = Cycle_Offset

with

Cycle_Repetition selected from the set of {1, 2, 4, 5, 8, 10, 16, 20, 32, 40, 50, 64}

Cycle_Offset selected from the set {0 … 63} with Cycle_Offset < Cycle_Repetition

In case the CHI allows that multiple buffers are configured for the same slot the CHI has to select a unique buffer for transmission in a deterministic way which is predictable by the host (see section 9.3.2.6).

Each transmit buffer is associated with a payload data valid flag and a transmission mode indicator.

The payload data valid flag denotes whether the message contained in the transmit buffer is valid or not.

For each transmit buffer the CHI shall ensure that the protocol engine either

1. is provided with a consistent set of valid message data from the transmit buffer, or
2. receives an indication that a consistent read of message data is not possible or that the transmit buffer contains no valid message (i.e. when the payload data valid flag is set to false).

The CHI shall support at least two modes, which determine how the payload data valid flag shall be updated after a transmission:

1. single shot transmission mode:
   When payload data has been provided by the host and marked as valid by setting the payload data valid flag to true, the data remains valid until the data has been transmitted (i.e., the protocol engine returns *vSS!FrameSent* as true). After the transmission, the CHI shall automatically invalidate the data by setting the payload data valid flag to false (i.e., the payload data is transmitted exactly once[152] as the result of the buffer update by the host).
2. continuous transmission mode:
   When payload data has been provided by the host and marked as valid by setting the payload data valid flag to true, the data remains valid until the host explicitly marks the data as invalid by setting the payload data valid flag to false (i.e., the payload data is transmitted repeatedly until the host invalidates the buffer data).

9.3.2.7.2 Transmit buffer identification for message retrieval

The protocol engine interacts with the controller host interface by importing the message data provided by the CHI at the start of each slot according to the media access control processes defined in Chapter 5. The protocol engine imports data elements from the CHI as defined in Definition 5-3. In response to each request the CHI shall perform the following steps:

[152] Note that in the static segment null frames may be transmitted when the payload data valid flag is false.

1. The CHI shall check whether the slot is assigned to the node on the relevant channel by querying the transmission slot assignment list for the relevant channel using *vCycleCounter* and the channel-specific value for *vSlotCounter*.
2. If the slot is not assigned to the node then the CHI shall return *vTCHI* with *vTCHI!Assignment* set to UNASSIGNED else *vTCHI!Assignment* shall be set to ASSIGNED and the subsequent steps shall be executed.
3. *vTCHI!HeaderCRC* shall be set to the value of the header CRC retrieved from the transmission slot assignment list.[153]
4. The active transmit message buffer shall be identified according to the process described in section 9.3.2.6.
5. If there is no active transmit message buffer then the CHI shall signal to the protocol engine that the communication slot is assigned but without any message data available by setting *vTCHI!TxMessageAvailable* to false and returning *vTCHI*.
6. If there is an active transmit message buffer then a consistent read of its data shall be attempted.
7. If a consistent read is not possible (for example, if the buffer is locked[154]), or the payload data valid flag is set to false, then the CHI shall signal to the protocol engine that the communication slot is assigned but without any message data available by setting *vTCHI!TxMessageAvailable* to false and returning *vTCHI*.
8. If a consistent read is possible, the CHI shall signal to the protocol engine that the message data is available for this communication slot by setting
   a. *vTCHI!TxMessageAvailable* to true,
   b. *vTCHI!PPIndicator* to the value retrieved from the transmit buffer configuration data as defined by the message ID filtering service in section 9.3.3.3 and the network management service in section 9.3.3.4,
   c. *vTCHI!Length* to the length of the message *MessageLength* held in the corresponding transmit buffer,
   d. *vTCHI!Message* to the message data from the transmit buffer

   and returning *vTCHI*.

9.3.2.7.3 Transmit buffer status

A message buffer configured for transmission shall be able to hold a snapshot of the following status information:

1. a frame transmitted indicator that shall be set to true if a frame that was not a null frame was completely transmitted[155]. The CHI shall set the frame transmitted indicator to true if there was a complete frame transmission in the slot (*vSS!FrameSent* is true), and shall leave the frame transmitted indicator at its current value if there was not a complete frame transmission in the slot (*vSS!FrameSent* is false). The CHI shall provide a mechanism which allows the host to reset the frame transmitted indicator.
2. a syntax error flag that shall be set if a syntax error was observed in the transmission slot (*vSS!SyntaxError* set to true) or cleared if no syntax error was observed in the transmission slot (*vSS!SyntaxError* set to false),

[153] Note that the header CRC information in the transmission slot assignment list may be derived dynamically from information in the transmit buffer configuration.

[154] See section 9.3.2.5.4. Note that in most circumstances a buffer could not be an active transmit message buffer if it is locked. It is possible, however, that the buffer is locked by the host after it has already been selected as the active transmit message buffer.

[155] A frame is considered completely transmitted at the start of transmission of the FES. See the *frame transmitted on A* signal in Figure 3-19 for details.

3. a content error flag that shall be set if a content error was observed in the transmission slot (*vSS!ContentError* set to true) or cleared if no content error was observed in the transmission slot (*vSS!ContentError* set to false),
4. a slot boundary violation flag that shall be set if a slot boundary violation, i.e. channel active at the start or at the end of the slot, was observed in the transmission slot (*vSS!BViolation* set to true) or cleared if no slot boundary violation was observed in the transmission slot (*vSS!BViolation* set to false),
5. a transmission conflict flag that shall be set if a transmission conflict error was observed in the transmission slot (*vSS!TxConflict* set to true) or cleared if no transmission conflict error was observed in the transmission slot (*vSS!TxConflict* set to false),
6. a valid frame flag that shall reflect the status of the *vSS!ValidFrame* variable in the transmission slot status.[156]

The slot status updated indicator shall be set to true by the CHI when the slot status has been updated. The CHI shall provide a mechanism that allows the host to reset this indicator.

If a transmit buffer is configured for both channel A and channel B it has to be capable of storing the above-listed status information separately for each channel, and it must be possible to determine the corresponding channel for each set of status information.

9.3.2.8 Message reception

Message reception operates on queued and/or non-queued receive buffers. A non-queued receive buffer is a data storage structure

1. for which the host has access to the data through a read operation,
2. for which the protocol engine has access to the data through a write operation, and
3. in which new values overwrite former values.

Refer to sections 9.3.2.8.1 and 9.3.2.8.2 for the requirements on non-queued receive buffers.

A queued receive buffer is a data storage structure

1. for which the host has access to the data through a read operation,
2. for which the protocol engine has access to the data through a write operation, and
3. in which new values are queued behind former values.

Refer to section 9.3.2.10 for the requirements on queued receive buffers / FIFO's.

9.3.2.8.1 Non-queued receive buffer configuration

For each slot on each channel the protocol engine provides a tuple of values to the CHI consisting of a slot status *vSS* of the communication slot in the current communication cycle and, if a semantically valid frame was received in this communication slot on this channel, the contents *vRF* of the first semantically valid frame. Section 9.3.2.6 describes how a specific receive buffer is selected based on this tuple.

In general, a receive buffer will be configured with the following:

1. a slot identifier configuration identifying a single slot identifier,
2. a channel configuration identifying a set of channels (A, B, or both A and B),
3. a cycle counter configuration identifying a set of communication cycles.

It shall be possible to configure a receive buffer for a single slot or for a set of slots sharing the same slot identifier in a configurable set of communication cycles.

[156] In most cases *vSS!ValidFrame* will be set to false if a transmit buffer was selected as the active buffer, but there are circumstances where it could be set to true even though a transmit buffer was selected (if, for example, transmission is prohibited because the MAC is in the KEYSLOTONLY mode or if the node is in the *POC:normal passive* state).

The configuration of the set of communication cycles shall support, at a minimum, defining sets of communication cycles which equal the sets which can be defined using a Cycle_Repetition and a Cycle_Offset to determine the configuration:

The receive buffer is configured for a reception slot if

vCycleCounter mod Cycle_Repetition = Cycle_Offset

with

Cycle_Repetition selected from the set of {1, 2, 4, 5, 8, 10, 16, 20, 32, 40, 50, 64}

Cycle_Offset selected from the set {0 … 63} with Cycle_Offset < Cycle_Repetition

For frames received in the static segment, the following channel configuration shall be supported:

1. receive buffer configured for channel A
2. receive buffer configured for channel B
3. for dual channel devices, receive buffer configured for both channel A and channel B. In this case the CHI shall select among the information provided by the channel specific FSP processes of the protocol engine. If the protocol engine provides only a single valid frame, that frame should be stored in the buffer, regardless of which channel it was received on. If the protocol engine provides two valid frames (one from each channel), and only one of the frames is a non-null frame, the non-null frame should be stored in the buffer, regardless of which channel this non-null frame was received on. If the protocol engine provides two valid frames and both are non-null frames[157], the receive buffer shall store the frame that was received on channel A[158].

Section 9.3.2.6 defines the requirements for resolving situations when more than one non-queued receive buffer can serve as a candidate buffer for a given slot / cycle / channel combination. This process can result in ambiguity in a case where one buffer is configured for channel A only (or channel B only, or channel A&B) while another buffer is configured for both channel A and channel B for the same (or an overlapping) slot / cycle combination. This specification makes no requirements on the buffer selection behavior of an implementation for such configurations - the behavior in these circumstances is implementation dependent.[159]

For frames received in the dynamic segment, the following channel configurations shall be supported:

1. receive buffer configured for channel A
2. receive buffer configured for channel B.

For each receive buffer the CHI shall ensure that the host either

1. is provided with a consistent set of message data from the receive buffer, or,
2. receives an indication that a consistent read of the message data is not possible.

For each receive buffer the CHI shall ensure that the information provided by the protocol engine is written to the corresponding receive buffer either

1. consistently, i.e. perform a consistent write of its data as if in one indivisible operation, or
2. not at all. In this case the payload data valid flag shall be set to false, so the host can assess that receive buffer contents were lost.

In case the CHI allows that multiple buffers are configured for the same slot the CHI has to select a unique buffer in a deterministic way which is predictable by the host (see section 9.3.2.6).

If for the same slot both a receive buffer and a transmit buffer are configured, by following the process described in section 9.3.2.6 the CHI will ensure that the content of the active transmit buffer is provided to the protocol engine. In such a case the transmission has priority over the reception.

[157] If both valid frames are null frames only the slot status from both channels is stored in the buffer.
[158] The preference for channel A is entirely arbitrary, serving only to define a deterministic behavior.
[159] The use of such a configuration is not recommended.

Each receive buffer shall hold up to a buffer specific bound of two-byte words.

For non-queued receive buffers this buffer specific bound may be set individually for each receive buffer within a node between 1 and *cPayloadLengthMax*.

9.3.2.8.2 Non-queued receive buffer contents

Each receive buffer shall contain slot status data as well as frame contents data.

9.3.2.8.2.1 Slot status data

A message buffer configured for reception shall be able to store a snapshot of the following slot status variables:

1. a valid frame flag that shall be set if a syntactically and semantically correct frame was received in the corresponding slot (*vSS!ValidFrame* set to true) and cleared if no valid frame was received (*vSS!ValidFrame* set to false),
2. a syntax error flag that shall be set if a syntax error was observed in the corresponding slot (*vSS!SyntaxError* set to true) or cleared if no syntax error was observed in the corresponding slot (*vSS!SyntaxError* set to false),
3. a content error flag that shall be set if a content error was observed in the corresponding slot (*vSS!ContentError* set to true) or cleared if no content error was observed in the corresponding slot (*vSS!ContentError* set to false),
4. a slot boundary violation flag that shall be set if a slot boundary violation, i.e. channel active at the start or at the end of the slot, was observed in the corresponding slot (*vSS!BViolation* set to true) or cleared if no slot boundary violation was observed in the corresponding slot (*vSS!BViolation* set to false),
5. a null frame indicator flag that shall be set according to the value of the *vSS!NFIndicator* status of the corresponding slot. If no valid frame was received the flag *vSS!NFIndicator* will be set to 0. This flag will only have a value of 1 if the slot contained a valid, non-null frame.

The slot status updated indicator shall be set to true for the message buffer by the CHI when the slot status has been updated. The CHI shall provide a mechanism that allows the host to reset this indicator.

If a receive buffer is configured for both channel A and channel B it has to be capable of storing the above-listed status information separately for each channel, and it must be possible to determine the corresponding channel for each set of status information.

Table 9-2 lists all possible combinations of *ValidFrame*, *SyntaxError*, *ContentError* and *BViolation* for the static and the dynamic segment along with a set of interpretations concerning the number of syntactically[160] and semantically[161] valid frames received in a static or dynamic slot, respectively.

Valid Frame	Syntax Error	Content Error	BVio-lation	Nothing was received (silence)	One or more syntactically valid frames were received	At least one semantically valid frame was received	Additional activity and a semantically valid frame was received
false	false	false	false	Yes	No	No	-
false	true	false	false	No	No	No	-
false	false	true	false	No	Yes	No	-

Table 9-2: Slot status interpretation.

[160] A frame is syntactically valid if it fulfills the decoding rules defined in section 3.3.5 including the check for a valid header CRC and a valid frame CRC in accordance with the number of two-byte payload words as denoted in the header of the frame.

Valid Frame	Syntax Error	Content Error	BVio-lation	Nothing was received (silence)	One or more syntactically valid frames were received	At least one semantically valid frame was received	Additional activity and a semantically valid frame was received
false	true	true	false	No	Yes	No	-
false	false	false	true	No	No	No	-
false	true	false	true	No	No	No	-
false	false	true	true	No	Yes	No	-
false	true	true	true	No	Yes	No	-
true	false	false	false	No	Yes	Yes	No
true	true	false	false	No	Yes	Yes	Yes[a]
true	false	true	false	No	Yes	Yes	Yes
true	true	true	false	No	Yes	Yes	Yes
true	false	false	true	No	Yes	Yes	Yes
true	true	false	true	No	Yes	Yes	Yes
true	false	true	true	No	Yes	Yes	Yes
true	true	true	true	No	Yes	Yes	Yes

Table 9-2: Slot status interpretation.

a. The syntax error indication may be caused by additional activity, but it could also be caused by a decoding error in the second bit of the FES of a frame otherwise free of decoding errors. In the latter case, there may or may not be additional activity.

9.3.2.8.2.2 Frame contents data

A message buffer configured for reception shall be able to store a snapshot of the following frame contents variables at the end of the communication slot if a valid frame was received in the slot (*vSS!ValidFrame* is equal to true) and the frame contained valid payload data (*vRF!Header!NFIndicator* is equal to one)[162] and the buffer was the active message buffer:

1. the reserved bit *vRF!Header!Reserved*,
2. the frame ID *vRF!Header!FrameID*,
3. the cycle counter *vRF!Header!CycleCount*,
4. the length field *vRF!Header!Length*,
5. the header CRC *vRF!Header!HeaderCRC*,
6. the payload preamble indicator *vRF!Header!PPIndicator*,
7. a flag that indicates that the buffer has been updated at some point during operation. This flag should be set to zero at buffer configuration, and be set to one when the buffer is updated as a result of the reception of a valid, non-null frame. Note that this behavior could be achieved by simply copying the *vRF!Header!NFIndicator* flag whenever the *vRF* structure is passed to the CHI from the protocol engine.

[161] A semantically valid frame is a syntactically valid frame that also fulfills a set of content related criteria.
[162] The reception of a null frame should not cause a snapshot of the frame contents data to be stored in the buffer.

8. The sync frame indicator *vRF!Header!SyFIndicator*,
9. the startup frame indicator *vRF!Header!SuFIndicator*,
10. *vRF!Header!Length* number of two-byte payload data words from *vRF!Payload*, if *vRF!Header!Length* does not exceed the buffer length *BufferLength* of the receive message buffer,
11. *BufferLength* number of two-byte payload data words from *vRF!Payload*, if *vRF!Header!Length* exceeds the buffer length *BufferLength* of the receive message buffer.[163]

In addition to the above, if a receive buffer is configured for both channel A and channel B it must also store information allowing the determination of the source of the frame contents data stored in the buffer:

12. the channel indicator *vRF!Channel*.

An exception to the normally required behavior exists during the first slot of the first cycle of operation of a TT-E coldstart node. Due to the short time between availability of information on the current cycle count and the start of the first cycle of operation it may not be possible for an implementation to complete a search of the entire set of buffers during the first slot. As a result, it may be possible that even though a buffer may be configured that an implementation may not be aware of this in time to behave as described in this specification. As a result, a specific exception is made for the first slot of the first cycle after a TT-E coldstart node's transition from the *POC:external startup* state to the *POC:normal active* state - such a node is allowed, but not required, to update the frame contents data for a receive buffer for this slot.

9.3.2.9 Non-queued message buffer status update

For a given channel the protocol engine makes the slot status *vSS* available to the CHI at the end of each slot.[164]

In case that there is an active message buffer available for the given channel the CHI shall copy a snapshot of the relevant *vSS* status information into the status data of the active message buffer (see sections 9.3.2.7.3 and 9.3.2.8.2.1).

The active message buffer is available for the status update when

- the active message buffer has been identified at the start of the slot

AND

- the host did not reconfigure the active message buffer between the time of the identification of the active message buffer and the point in time the status update should occur

AND

- the active message buffer is not locked at the time the status update should occur (see section 9.3.2.5.4)

AND in case of an active transmit message buffer

- there has been no write access by the host to the message buffer between the time the payload is provided to the protocol engine and the point in time at which the status update should occur.

If an active message buffer is configured for both channels, only the portion of the status relevant for the channel(s) for which the active message buffer has been identified will be updated[165].

In case there is no active message buffer available for the given channel at the time of the slot status update no non-queued message buffer will be updated with status information.

---

[163] The host can assess such a truncation through the data element *vRF!Header!Length*.

[164] During the startup phase no buffer update will take place.

[165] Under certain configurations this could result in only one of the two sets of channel-specific status information being updated. This could happen, for example, if the configuration allows a buffer configured for both channels to be selected as the active buffer on one channel and not the active buffer on the other channel. Since there is only one slot status updated flag for a buffer, the host would not be able to determine that such a "half update" took place. As a result, configurations in which this could occur should be avoided.

An exception to the normally required behavior exists during the first slot of the first cycle of operation of a TT-E coldstart node. Due to the short time between availability of information on the current cycle count and the start of the first cycle of operation it may not be possible for an implementation to complete a search of the entire set of buffers during the first slot. As a result, it may be possible that even though a buffer may be configured that an implementation may not be aware of this in time to behave as described in this specification. As a result, a specific exception is made for the first slot of the first cycle after a TT-E coldstart node's transition from the *POC:external startup* state to the *POC:normal active* state - such a node is allowed, but not required, to update the slot status information for a receive or a transmit buffer for this slot.

9.3.2.10 Queued receive buffers (FIFO's)

Queued receive buffers, also referred to as FIFO buffers, are a class of receive buffer that are capable of storing status information and payload data for more than one frame. Information is placed into the FIFO by the CHI and is removed from the FIFO by the host. The FIFO represents a queue, i.e., it is possible that multiple messages go into the FIFO before any of the messages are removed by the host, and it is possible that the host reads multiple messages that have already been previously placed in the FIFO even though no additional messages have been placed into the FIFO.

The FlexRay FIFO preserves the order of the messages which are placed into the FIFO - messages are removed from the FIFO in the same order they are placed into the FIFO (i.e., it has a "First In First Out" behavior). Refer to section 9.3.2.10.3 for additional details.

When the host removes a message from the FIFO the removal frees up FIFO resources that can be used to store additional messages in the FIFO. As long as the host removes the entries from the FIFO often enough that the FIFO does not fill up, all messages that should go into the FIFO are made available to the host[166].

9.3.2.10.1 Basic FIFO behavior

A FIFO buffer consists of a number of entries, each of which is capable of storing information related to the reception of a frame. Status variables and frame contents data are only stored in a FIFO buffer when the admittance criteria are passed (see 9.3.2.10.2) and the FIFO buffer is selected (see 9.3.2.10.1.1). Note that unlike non-queued receive buffers there is no requirement to store slot status information for a slot if no valid frame was received.

A FIFO buffer shall be able to store a snapshot of the following slot status variables when a frame is admitted into the FIFO:

1. a syntax error flag that shall be set if a syntax error was observed in the corresponding slot (*vSS!SyntaxError* set to true) or cleared if no syntax error was observed in the corresponding slot (*vSS!SyntaxError* set to false),
2. a content error flag that shall be set if a content error was observed in the corresponding slot (*vSS!ContentError* set to true) or cleared if no content error was observed in the corresponding slot (*vSS!ContentError* set to false),
3. a slot boundary violation flag that shall be set if a slot boundary violation, i.e. channel active at the start or at the end of the slot, was observed in the corresponding slot (*vSS!BViolation* set to true) or cleared if no slot boundary violation was observed in the corresponding slot (*vSS!BViolation* set to false),
4. a null frame indicator flag that shall be set according to the value of the *vSS!NFIndicator* status of the corresponding slot. If no valid frame was received the flag *vSS!NFIndicator* will be set to 0. This flag will only have a value of 1 if the slot contained a valid, non-null frame.

A FIFO buffer shall be able to store a snapshot of the following frame contents variables when a frame is admitted into the FIFO:

[166] If the host does not remove the messages from the FIFO often enough eventually the FIFO will reach its limits and an overrun will occur. The behavior of the FIFO in this case is implementation dependent.

1. the reserved bit *vRF!Header!Reserved*,
2. the frame ID *vRF!Header!FrameID*,
3. the cycle counter *vRF!Header!CycleCount*,
4. the length field *vRF!Header!Length*,
5. the header CRC *vRF!Header!HeaderCRC*,
6. the payload preamble indicator *vRF!Header!PPIndicator*,
7. The sync frame indicator *vRF!Header!SyFIndicator*,
8. the startup frame indicator *vRF!Header!SuFIndicator*,
9. *vRF!Header!Length* number of two-byte payload data words from *vRF!Payload*, if *vRF!Header!Length* does not exceed the width of the selected FIFO buffer (see section 9.3.2.10.3),
10. a number of two-byte words equal to the width of the selected FIFO buffer from *vRF!Payload*, if *vRF!Header!Length* exceeds the width of the selected FIFO buffer.

In addition to the above, if a FIFO may be configured to admit frames from both channel A and channel B it must also be capable of storing information allowing the determination of the source of the frame contents data stored in a FIFO entry:

11. the channel indicator *vRF!Channel*.

For each FIFO buffer entry the CHI shall ensure that the host either

1. is provided with a consistent set of message data from the receive buffer, or,
2. receives an indication that a consistent read of the message data is not possible.

For each FIFO buffer entry the CHI shall ensure that the data provided by the protocol engine is written to the corresponding FIFO buffer either

1. consistently, i.e. perform a consistent write of its data in one indivisible operation, or
2. not at all. In this case a flag shall be provided through which the host can assess that receive buffer contents were lost.

9.3.2.10.1.1 Admittance into a FIFO

A FIFO buffer has a set of admittance criteria (that determines when the CHI puts frame status and payload data from the protocol engine into the FIFO). When a frame matches all of the admittance criteria (refer to section 9.3.2.10.2) it is placed into the FIFO.

A frame is only considered for admittance into a FIFO if no match for the frame is found within the configured non-queued receive buffers (i.e., the FIFO's have lower priority than the non-queued receive buffers described in section 9.3.2.8.1).

If the CHI supports multiple FIFO's and if the admittance criteria for the FIFO's are configurable such that a single frame can meet the admittance criteria of more than one FIFO the CHI shall select a unique FIFO buffer in a deterministic, implementation dependent manner.

The host shall have read access to admittance criteria independent of the protocol state.

The host shall have write access to the admittance criteria in the *POC:config* state.

Enabling and disabling the write access to the admittance criteria during *POC:normal active* state and *POC:normal passive* state shall only be possible in the *POC:config* state.

Host write access to the admittance criteria shall be possible in *POC:normal active* state and in *POC:normal passive* state, if this was explicitly enabled during *POC:config* state.

When the host is changing the set of admittance criteria in the *POC:normal active* state or in the *POC:normal passive* state, the CHI shall prevent that an inconsistent set of admittance criteria (i.e., the changes are incomplete) is applied.

After changing the admittance criteria in the *POC:normal active* state or *POC:normal passive* state, the CHI shall continue to provide host access to the existing entries in the FIFO. New entries shall be placed into the FIFO according to the new admittance criteria.

An exception to the normally required behavior exists during the first slot of the first cycle of operation of a TT-E coldstart node. Due to the short time between availability of information on the current cycle count and the start of the first cycle of operation it may not be possible for an implementation to complete a search of the entire set of non-queued buffers during the first slot. As a result, it may be possible that even though a non-queued buffer is configured that an implementation may not be aware of this in time to behave as described in this specification. Further, it may not be possible for an implementation that implements the majority of the FIFO admittance checks prior to the start of a slot to complete all checks in time to process a frame received in the first slot of operation. As a result, a specific exception is made for the first slot of the first cycle after a TT-E coldstart node's transition from the *POC:external startup* state to the *POC:normal active* state - such a node is allowed, but not required, to consider a received frame for admittance into a FIFO buffer. If an implementation does consider a frame for admission it shall meet all of the requirements described in this section, specifically, that there be no match for the frame within the configured non-queued receive buffers.[167]

9.3.2.10.1.2 Reading and removal from a FIFO

The CHI shall provide a method to allow the host to read the slot status data and frame contents data stored in the first (oldest) FIFO entry. In addition, the CHI shall provide a mechanism to allow the host to remove the first entry from the FIFO without requiring the host to read the entire FIFO entry.

It shall be possible for the host to read any portion of the slot status data and / or the frame contents data of the first entry in the FIFO and then make a decision as to whether or not to read the remaining data in the FIFO entry and still have access to the remaining data, i.e., it is not acceptable for a read of any portion of the data alone to cause data to be lost[168].

It shall be possible for the host to read information out of the FIFO (and remove messages from the FIFO) at that same time that other frames are being placed into the FIFO.

Upon a transition from either the *POC:normal active* or *POC:normal passive* state to either the *POC:halt* or *POC:ready* state the CHI shall continue to provide host access to the FIFO buffer entries that it would have provided had the POC remained in the *POC:normal active* or *POC:normal passive* states.

The behavior of the CHI upon attempted host access to queued buffer status or payload data that has never been updated is implementation dependent. It is required, however, that the access to such data does not give the appearance that data was received when such a reception did not actually take place.

9.3.2.10.2 FIFO admittance criteria

An implementation shall be capable of determining which frames will be placed into a FIFO structure. The decision as to whether or not a frame is placed into a FIFO buffer is based on a series of five admittance criteria:

1. FIFO frame validity admittance criteria
2. FIFO channel admittance criteria
3. FIFO frame identifier admittance criteria
4. FIFO cycle counter admittance criteria
5. FIFO message identifier admittance criteria

[167] As a result, an implementation that is not able to search all non-queued buffers is not allowed to consider the frame for admittance into a FIFO buffer.

[168] In this context being "lost" is different from being removed from the FIFO - depending on the implementation, it may be possible to remove an entry from the FIFO without the data being lost.

Each received frame for which there is no active non-queued buffer is a candidate for admission into a FIFO received buffer.[169] Each candidate frame is checked against the FIFO admittance criteria to determine if is placed into a FIFO receive buffer.

A frame must pass all five of the admittance criteria in order to be placed into a FIFO buffer - if one or more of the admittance criteria fail the frame will not be placed into a FIFO buffer.

An implementation shall support the admittance criteria (and configuration of the admittance criteria) as described in the following sections.

9.3.2.10.2.1 FIFO frame validity admittance criteria

An implementation shall be capable of admitting or not admitting a frame into the FIFO based on the validity of the frame. In addition, an implementation shall be capable of being configured to admit or not admit valid null frames into the FIFO. Specifically, the frame validity admittance criteria shall have a single Boolean configuration, AdmitNullFrame.

The frame validity admittance criteria shall have the following behavior:

if AdmitNullFrame = false then all valid non-null frames (i.e., frames that cause an FSP process to generate an 'update *vRF* on A in CHI' or 'update *vRF* on B in CHI' - refer to Figure 6-9) shall be considered to pass the FIFO frame validity admittance criteria. All other frames (or other activity) shall be considered to fail the frame validity admittance criteria.

if AdmitNullFrame = true, then all valid frames (*vSS!ValidFrame* is equal to true) shall be considered to pass the FIFO frame validity admittance criteria[170]. All other frames (or other activity) shall be considered to fail the frame validity admittance criteria.

9.3.2.10.2.2 FIFO channel admittance criteria

An implementation shall be capable of being configured to admit or not admit a frame into the FIFO based on the channel on which the frame was received. Specifically, an implementation must be capable of configuring the FIFO channel admittance criteria such that a frame shall be considered to pass this admittance criteria

- only if the frame was received on Channel A, or
- only if the frame was received on Channel B, or
- regardless of whether the frame was received on Channel A or Channel B[171]

If the FIFO channel admittance criteria is configured for the last option (i.e., to admit frames regardless of channel), and for a given slot frames are received on both channel A and channel B, and both frames pass all of the other admittance criteria, then both frames must be accepted into a FIFO structure[172].

[169] An interesting situation can arise if a non-queued buffer is configured to receive a frame on both channels, and valid frames actually occur on both channels. In this case, even though the payload information for only one of the frames can be stored in the receive buffer (see section 9.3.2.8.1), both frames actually have an active non-queued buffer (for example, the active buffer will store slot status information affected by the frames on both channels). As a result, neither frame would be a candidate for admission into a FIFO receive buffer.

[170] Note that these criteria include all frames that would be admitted when AdmitNullFrame = false, but also include valid null frames.

[171] It is required that dual channel implementations be able to be configured to support the reception of frames on either channel A or channel B into some FIFO structure, but if more than one FIFO structure is available, then it is not necessary that frames from channel A and channel B be received into the same FIFO structure. See section 9.3.2.10.3 for further information.

[172] Note that the required behavior of the FIFO differs from the optional behavior of receive buffers in the static segment defined in section 9.3.2.8.1. If the admittance criteria is set for both channel A and channel B, then both frames, rather than just the first valid non-null frame, must be entered into a FIFO structure.

9.3.2.10.2.3 FIFO frame identifier admittance criteria

An implementation shall be capable of being configured to admit or not admit a frame into the FIFO based on the Frame ID[173] of the received frame. Specifically, the FIFO frame identifier admittance criteria shall be considered to be passed only if the received frame identifier belongs to a configurable set of frame identifiers referred to as the FIFO frame identifier set. If the received frame identifier is not a member of the FIFO frame identifier set the FIFO frame identifier admittance criteria shall be considered to be failed.

At a minimum, the FIFO frame identifier set shall be capable of being configured to any set that could be configured with the following abstract definition[174]:

Define four parameters, $Range1_{min}$, $Range1_{max}$, $Range2_{min}$, and $Range2_{max}$, each configurable in the range of {0 … *cSlotIDMax*}. The FIFO frame identifier set is the set of all frame identifiers Frame ID in the range {1 … *cSlotIDMax*} such that

$Range1_{min}$ <= Frame ID <= $Range1_{max}$

or

$Range2_{min}$ <= Frame ID <= $Range2_{max}$

Note that the individual ranges identified above can cross the boundary between the static and dynamic segment, i.e., $Range1_{min}$ could be an identifier within the static segment and $Range1_{max}$ could be an identifier within the dynamic segment. The FIFO frame identifier definition above applies even if Frame ID lies in a different segment than the minimum or maximum value of the range configuration.

9.3.2.10.2.4 FIFO cycle counter admittance criteria

An implementation shall be capable of being configured to admit or not admit a frame into the FIFO based on the cycle counter value at the time the frame is received. Specifically, the FIFO cycle counter admittance criteria shall be considered to be passed only if the cycle counter value when the frame was received belongs to a configurable set of cycle counter values referred to as the FIFO cycle counter set. If the cycle counter when the frame was received is not a member of the FIFO cycle counter value set the FIFO cycle counter value admittance criteria shall be considered to be failed.

At a minimum, the FIFO cycle counter set shall be capable of being configured to any set that could be configured with the following abstract definition[175]:

Define two parameters, Cycle_Repetition and Cycle_Offset, with

Cycle_Repetition selected from the set of {1, 2, 4, 5, 8, 10, 16, 20, 32, 40, 50, 64}

Cycle_Offset selected from the set {0 … 63}

Cycle_Offset < Cycle_Repetition

The FIFO cycle counter value set is the set of all cycle counter values Cycle_Counter in the range of {0 … 63} such that

vCycleCounter mod Cycle_Repetition = Cycle_Offset

[173] This admittance criterion is actually based on the current value of the slot counter at the time the frame is received rather than on the Frame ID received in the frame. Since the frame validity admittance criteria will only admit valid frames into the FIFO, the only frames that could be admitted in the FIFO are those whose received frame identifier matches the current slot counter. The term frame identifier is used in this section for clarity only - strictly speaking, the implementation would be based on the slot counter at the time of frame reception.

[174] Note that an implementation does not need to explicitly support the parameters in the abstract definition, but needs to be able to generate all of the FIFO frame identifier sets that could be generated by the abstract definition.

[175] Note that an implementation does not need to explicitly support the parameters in the abstract definition, but needs to be able to generate all of the FIFO cycle counter sets that could be generated by the abstract definition.

9.3.2.10.2.5 Message identifier admittance criteria

An implementation shall be capable of being configured to admit or not admit a frame into the FIFO based on the value of the optional Message ID that can be present in frames received in the dynamic segment. Message identifier admittance criteria shall have the following characteristics:

The message identifier admittance criteria shall have three configurable parameters[176]:

1. MsgIDMask, an integer in the range of {0 … 65535}, which determines the bitwise AND mask value for the admittance criteria
2. MsgIDMatch, an integer in the range of {0 … 65535}, which determines the bitwise comparison value for the admittance criteria
3. AdmitWithoutMessageID, a Boolean configuration which determines whether or not frames received in the dynamic segment that don't contain a message ID will be admitted into the FIFO.

The message identifier admittance criteria shall have the following behavior:

- If the frame was received in the static segment the frame is considered to have passed the message identifier admittance criteria, or
- If the frame was received in the dynamic segment and does not contain a message identifier (i.e., the payload preamble indicator of the frame is set to zero), the behavior depends on the AdmitWithoutMessageID configuration of the FIFO:

  If AdmitWithoutMessageID = false the frame is considered to fail the message identifier admittance criteria.

  If AdmitWithoutMessageID = true the frame is considered to pass the message identifier admittance criteria.

  or
- If the frame was received in the dynamic segment and does contain a message identifier (i.e., the payload preamble indicator of the frame is set to one), the admittance depends on the value of the message identifier, MessageID, and the FIFO configurations MsgIDMask and MsgIDMatch. Specifically, the frame is considered to pass the message identifier admittance criteria if

  (MessageID & MsgIDMask) = MsgIDMatch

  otherwise the frame is considered to fail the message identifier admittance criteria. Here the "&" symbol represents a bitwise AND of the binary representation of the values of MessageID and MsgIDMask[177].

9.3.2.10.3 FIFO performance requirements

An implementation shall provide at least one FIFO receive buffer structure.

A FIFO receive buffer structure shall have a depth of at least eight entries (i.e., in the absence of entries being removed from the FIFO by the host, the FIFO shall be capable of storing at least eight frames from the protocol engine without any loss or overwrite of frame contents data or slot status data).

Each entry in a FIFO receive structure also has a "width", i.e., each entry in a FIFO shall be capable of storing a number of bytes of frame payload data greater than or equal to the implementation's capability to store payload data for non-queued receive buffers. If a valid message is received whose payload is longer than the configured width of a FIFO entry, or is longer than the implementation's maximum width for FIFO's,

[176] Note that unlike other FIFO admittance criteria, an implementation must explicitly support the specified parameters of the message ID admittance criteria, i.e., it is not acceptable to support other means of configuring this admittance criteria.

[177] For example, if MsgIDMask = 3855 (0x0F0F hex) and MsgIDMatch = 1537 (0x0601 hex), then a frame with a MessageID of 38641 (0x96F1 hex) would pass the admittance criteria, but a frame with MessageID 1538 (0x0602 hex) would fail the criteria. Note that a configuration with MsgIDMask = 0 and MsgIDMatch = 0 would pass this criteria for any frame regardless of message identifier.

the implementation shall store the message's first payload bytes up to the configured or maximum length in the FIFO entry.

At a minimum an implementation shall provide the capability to store at least eight entries with the width defined above (i.e., the depth and width requirements must be met simultaneously).

An implementation that supports two channels must be capable of receiving frames from each channel into some FIFO structure[178].

The requirements on buffer depth and width, FIFO admittance criteria, etc. are channel independent (i.e., the FIFO's of an implementation need to be able to support the indicated number of frames entirely on channel A, entirely on channel B, or any mixture of channel A and B[179]). Note there is no requirement that an implementation needs to support the indicated number of messages on both channel A and Channel B at the same time, or that the FIFO admittance criteria are able to be independently set for Channel A and Channel B.

A dual channel implementation must be able to support simultaneous reception on channel A and B (i.e., if frames appear on both channels at the same time, or with arbitrary overlap, and both frames meet the admittance criteria for the FIFO, then both must be entered into some FIFO receive buffer structure).

An implementation must place frames into a FIFO in the order that they were received (i.e., a frame that was received earlier would be removed from a FIFO before a frame that was received later). There is no requirement for an implementation to maintain the relative order of frames received into different FIFO's, but it is required that a FIFO preserve the relative order of all messages received into the same FIFO.

9.3.2.10.4 FIFO status information

The following information on the status of a FIFO receive buffer structure shall be provided in the CHI:

1. The number of occupied entries currently in the FIFO
2. An overrun indicator is set if a FIFO overrun condition has occurred. An overrun occurs when a frame matches all of the FIFO admittance criteria but the FIFO is not capable of increasing the total number of entries in the FIFO. The overrun indicator shall remain set until explicitly cleared by the host. The behavior of the FIFO upon the occurrence of an overflow condition, and in recovery from an overflow condition, is implementation dependent.
3. Information that allows the host to determine how much of the FIFO remains available to accept additional information. The form of this information depends on the structure of the FIFO and is implementation dependent[180].

In addition to the previous status information, an implementation shall provide the ability to notify the host via an interrupt request when the available resources of the FIFO have fallen below a configurable level. The nature of the configurability is dependent on the structure of the FIFO and is implementation dependent[181].

[178] The intention is to allow implementations that dedicate separate FIFO's for each channel as well as implementations that support a single FIFO that can receive messages from either channel.

[179] For example, a dual channel implementation must be able support the reception of eight frames from channel A alone, or eight frames from channel B alone, seven from channel A and one from channel B, six from channel A and two from channel B, or any combination that adds up to eight frames total.

[180] For fixed size FIFO's with a fixed number of entries the number of messages currently in the FIFO would meet this requirement. For FIFO's with variable size structures some other implementation dependent mechanism to determine the amount of the FIFO resources available must be provided.

[181] For example, a FIFO that offers a fixed number of fixed size entries might be configurable based on the number of entries in the FIFO. A FIFO based on variable size entries might be configurable based on the fraction of the storage space remaining in the FIFO.

9.3.3 CHI Services

9.3.3.1 Macrotick timer service

The CHI shall provide at least two absolute timers capable of notifying the host at expiration.

Each of the required timers shall, at a minimum, be capable of being configured with the following expiration criteria:

- To expire at an absolute time in terms of cycle count and macrotick, i.e. the timer would expire at a configurable macrotick in a configurable communication cycle, and
- To expire at a configurable macrotick only (i.e., the timer would expire at a configurable macrotick independent of cycle count)

In addition, each of the required timers shall, at a minimum, be capable of supporting the following two modes of operation:

- A non-repetitive mode of operation where the timer will expire once the configured expiration criteria occurs and will not expire again until restarted or reconfigured by the host, and
- A repetitive mode of operation where the timer will expire every time the configured expiration criteria occurs (i.e., the timer can expire multiple times without further interaction from the host).

It shall be possible to configure and activate a timer when the protocol is in either the *POC:normal active* state or the *POC:normal passive* state.[182]

All absolute timers shall be deactivated when the protocol leaves the *POC:normal active* state or the *POC:normal passive* state apart from transitions between the *POC:normal active* state and the *POC:normal passive* state. It shall also be possible for the host to deactivate an absolute timer in any state that allows the timer to be activated. Once deactivated, a timer must be explicitly activated again by the host before it can expire again.

9.3.3.2 Interrupt service

The interrupt service provides a set of configurable interrupt requests to the host based on a set of interrupt sources reflecting events that occur in the protocol engine or the CHI.

At a minimum, an implementation shall provide the following interrupt sources:

- Each timer (as described in section 9.3.3.1) provided by an implementation shall be able to act as an interrupt source with the event being the expiration of the timer.
- The FlexRay cycle shall act as an interrupt source with the event being the start of a FlexRay cycle. This requirement is in addition to the previous timer requirement (i.e., it is not acceptable to meet this requirement through the use of a timer as specified in section 9.3.3.1).
- State transitions of the Protocol Operation Control process shall act as an interrupt source. This source shall only signal an event whenever the POC transitions to the *POC:halt*, *POC:ready*, *POC:normal active* or *POC:normal passive* states for any reason other than the processing of an IMMEDIATE_READY or FREEZE command.
- Each FIFO structure shall act as an interrupt source with the event being a frame reception that causes the available resources of the FIFO structure fall below a configurable level (refer to section 9.3.2.10.4).

An implementation shall provide a mechanism that allows the events of interrupt sources to generate interrupt requests (i.e., maps interrupt source events to interrupt requests). In general, a single interrupt request is allowed to support more than one interrupt source, however at least one of the mandated

[182] It is allowed, but not required, for an implementation to support configuration/activation of an absolute timer in states other than *POC:normal active* and *POC:normal passive*.

absolute timers (see section 9.3.3.1) shall provide a dedicated interrupt request that is not shared with any other interrupt source.

It shall be possible for the host to enable and disable the generation of interrupt requests for interrupt sources. When interrupt request generation is enabled for an interrupt source the event defined for the interrupt source shall cause the generation of a corresponding interrupt request. When interrupt request generation is disabled for an interrupt source the event defined for the interrupt source shall not cause the generation of a corresponding interrupt request.

An implementation shall provide control of interrupt request generation with at least two levels. At the first level, it shall be possible for the host to individually control whether interrupt request generation is enabled or disabled for each interrupt source. At the second level, it shall be possible for the host to globally disable all interrupt request generation regardless of the individual interrupt request generation enabled / disabled status of each interrupt source. Specifically, if interrupt request generation is globally disabled no interrupt source will generate interrupt requests; if interrupt request generation is not globally disabled each interrupt source shall generate interrupt requests according to its individual interrupt request generation status.

An implementation shall provide an interrupt status indication for each interrupt source. The interrupt status indication for an interrupt source shall be set when the interrupt source's event occurs[183] and shall remain set until reset under control of the host.

9.3.3.3 Message ID filtering service

The message ID filtering service provides means for selecting receive buffers based on a message ID that may be exchanged in the first two bytes of the payload segment of selected frames within the dynamic segment that have the payload preamble indicator set to one in the header of the frame.

To support this service the message buffer configuration data shall allow the host to configure the payload preamble indicator for each transmit buffer so that the host can configure whether the message data contains a message ID or not (refer to section 9.3.2.5.1).

Message ID filtering is required for at least one FIFO receive buffer structure as described in sections 9.3.2.10.2.5 and 9.3.2.10.3.

9.3.3.4 Network management service

The network management service provides means for exchanging and processing network management data. This service supports high-level host-based network management protocols that provide cluster-wide coordination of shutdown decisions based on the actual application state. The network management service may also be used by applications to implement other functionality.

Network management is performed by exchanging a network management vector in selected network management enabled frames within the static segment of the communication cycle that have the payload preamble indicator set to one in the header of the frame. The payload preamble indicator of the message buffer configuration data allows the host to configure whether or not a message contains a network management vector.

Throughout each communication cycle the CHI shall maintain an accrued network management vector by applying a bit-wise OR between the current accrued network management vector and each[184] network management vector received in a valid frame on each channel (regardless of whether or not any receive buffer is configured to explicitly receive the valid frame).[185]

---

[183] The interrupt status indication of an interrupt source is set whenever the corresponding event occurs irrespective of whether interrupt request generation is individually enabled for the interrupt source or whether interrupt request generation is globally disabled.

[184] Only valid frames reported to the CHI (via the *vRF* structures of the channel-specific FSP processes) are considered. If more than one frame occurs in a slot on a given channel only the first valid frame (i.e., the one reported in *vRF*) is considered.

1. If *pNMVectorEarlyUpdate* is set to false, the protocol status data shall contain a snapshot of the accrued network management vector that shall be updated no sooner than the end of the cycle and no later than the availability of the NIT status information or the availability of the payload data from the last static slot, whichever occurs later. The snapshot shall contain the value of the accrued network management vector at the end of the cycle, and shall include the effects of any reception that may have occurred in the last static slot.
2. If *pNMVectorEarlyUpdate* is set to true, the protocol status data shall contain a snapshot of the accrued network management vector that shall be updated no sooner than the end of the static segment and no later than the availability of the payload data from the last static slot. The snapshot shall contain the value of the accrued network management vector at the end of the static segment (i.e., shall include the effects of any reception that may have occurred in the last static slot).
3. These updates take place as long as the protocol is in either the *POC:normal active* state or the *POC:normal passive* state.
4. The accrued network management vector is set to zero at the beginning of each communication cycle.[186]
5. Following the completion of startup, the NM vector snapshot shall be set to all zeros prior to the first update of the NM vector snapshot (when this occurs depends on the configuration parameter *pNMVectorEarlyUpdate*).

In addition to the capabilities provided above, it is also possible for implementers to provide additional types of Network Management Services, for example, providing direct (non-OR'd) access to the NM Vector data.

[185] In this context, a frame is only considered valid if *vSS!ValidFrame* status is set to true when the protocol engine exports the slot status *vSS* to the CHI at the end of a slot or segment. For example, frames that are received in a slot which is also used for transmission shall not be accrued into the network management vector since such frames will have *vSS!ValidFrame* set to false when the slot status is exported to the CHI.

[186] Even though the snapshot of the NM vector described in item 1 might be presented to the CHI after the end of the cycle, it must represent the status of the accrued NM vector from the previous cycle. As a result, the snapshot must be taken before the vector is set to zero at the beginning of the subsequent cycle.

Appendix A System Parameters

A.1 Protocol constants

Name	Description	Value
cChannelIdleDelimiter	Duration of the channel idle delimiter.	11 gdBit
cClockDeviationMax	Maximum clock frequency deviation, equivalent to 1500 ppm (1500ppm = 1500/1000000 = 0.0015).	0.0015
cCrcInit[A]	Initialization vector for the calculation of the frame CRC on channel A (hexadecimal).	0xFEDCBA
cCrcInit[B]	Initialization vector for the calculation of the frame CRC on channel B (hexadecimal).	0xABCDEF
cCrcPolynomial	Frame CRC polynomial (hexadecimal).	0x5D6DCB
cCrcSize	Size of the frame CRC calculation register.	24 bits
cCycleCountMax	Maximum cycle counter value in any cluster.	63
cdBSS	Duration of the Byte Start Sequence.	2 gdBit
cdCAS	Duration of the logical low portion of the collision avoidance symbol (CAS) and media access test symbol (MTS).	30 gdBit
cdCASActionPointOffset	Initialization value of the CAS action point offset timer.	1 MT
cdCASRxLowMin	Lower limit of the CAS acceptance window.	29 gdBit
cdCycleMax	Maximum cycle length.	16000 µs
cdCycleStartTimeout	Maximum allowed jitter between the 'external cycle start' from the time gateway source and the internal 'cycle start' of the time gateway sink.	5 µT
cdFES	Duration of the Frame End Sequence.	2 gdBit
cdFSS	Duration of the Frame Start Sequence.	1 gdBit
cdInternalRxDelayMax	Maximum value of the implementation specific delay on the receive path of the decoder.	4 samples[a]
cdInternalRxDelayMin	Minimum value of the implementation specific delay on the receive path of the decoder.	1 sample[a]
cdMaxMTNom[b]	Maximum duration of a nominal macrotick. An implementation must be able to support nominal macrotick durations between *cdMinMTNom* and *cdMaxMTNom*.	6 µs

Table A-1: Protocol constants.

Name	Description	Value
cdMinMTNom[c]	Minimum duration of a nominal macrotick. An implementation must be able to support nominal macrotick durations between *cdMinMTNom* and *cdMaxMTNom*.	1 µs
cdStaggerDelay	Delay used to stagger the deactivation of the TxD and TxEN outputs during CAS/MTS and WUP transmission to eliminate the possibility of brief glitches. The TxD output will remain LOW for *cdStaggerDelay* following the deactivation of TxEN.	1 gdBit
cdTSrcCycleOffset	The delay between the cycle starts of the time gateway source and time gateway sink for a cluster operating in TT-E external sync mode.	40 µT
cdWakeupMaxCollision	Number of continuous bit times at LOW during the idle phase of a WUS that will cause a sending node to detect a wakeup collision.	5 gdBit
cdWakeupTxActive	Duration of the LOW phase of a transmitted wakeup symbol and the active (i.e., HIGH or LOW) phases of a transmitted WUDOP.	6 µs
cdWakeupTxIdle	Duration of the idle phase between two low phases inside a wakeup pattern.	18 µs
cFrameThreshold	Threshold used to differentiate noise from activity arising from a frame in the dynamic segment media access. Activity exceeding this threshold is assumed to have come from frame transmission as opposed to noise.	80 gdBit
cHCrcInit	Initialization vector for the calculation of the header CRC on channel A or channel B (hexadecimal).	0x01A
cHCrcPolynomial	Header CRC polynomial (hexadecimal).	0x385
cHCrcSize	Size of header CRC calculation register.	11 bits
cMicroPerMacroMin	Minimum number of microticks per macrotick during the offset correction phase.	20 µT
cMicroPerMacroNomMin	Minimum number of microticks in a nominal (uncorrected) macrotick.	40 µT
cMicroPerMacroNomMax	Maximum number of microticks in a nominal (uncorrected) macrotick.	240 µT
cPayloadLengthMax	Maximum length of the payload segment of a frame.	127 two-byte words
cPropagationDelayMax	Maximum allowable propagation delay arising from the physical layer and analog effects inherent in the FlexRay CC's involved in the transmission and reception of a communication element. These are the delays that occur between the points labeled as TP1_FF and TP4_FF in Figure 6-1 of [EPL10].	2.5 µs

Table A-1: Protocol constants.

Name	Description	Value
cSamplesPerBit	Number of samples taken in the determination of a bit value.	8
cSlotIDMax	Highest slot ID number.	2047
cStaticSlotIDMax	Highest static slot ID number.	1023
cStrobeOffset	Sample where bit strobing is performed (first sample of a bit is considered as sample 1).	5
cSyncFrameIDCountMax	Maximum number of distinct sync frame identifiers that may be present in any cluster.	15
cVotingDelay	Number of samples of delay between the RxD input and the majority voted output in the glitch-free case.	(*cVotingSamples* -1) / 2
cVotingSamples	Numbers of samples in the voting window used for majority voting of the RxD input.	5

Table A-1: Protocol constants.

a. This value is based on the experience of the semiconductor manufacturers of the FlexRay consortium.
b. This parameter is only introduced to be able to define a minimum conformance class range that all implementations must support. Note that this macrotick duration may not be achievable for all microtick durations - see section B.4.4 for details.
c. This parameter is only introduced to be able to define a minimum conformance class range that all implementations must support. Note that this macrotick duration may not be achievable for all bit rates or microtick durations - see section B.2 and B.4.4 for details.

A.2 Performance constants

Name	Description	Value
cdMaxOffsetCalculation	Maximum time allowed for calculation of the offset correction value, measured from the end of the static segment. In some situations the offset correction calculation deadline is actually longer - see section 8.6.2 for details.	1350 µT
cdMaxRateCalculation	Maximum time allowed for calculation of the rate correction value, measured from the end of the static segment. In some situations the rate correction calculation deadline is actually longer - see section 8.6.3 for details.	1500 µT

Table A-2: Performance constants.

Appendix B Configuration Constraints

B.1 General

This appendix specifies the configurable parameters of the FlexRay protocol. This appendix also identifies the configurable range of the parameters, and gives constraints on the values that the parameters may take on. All implementations that support a given parameter must support at least the parameter range identified in this appendix. An implementation is allowed, however, to support a broader range of configuration values.

Following functions are used for the configuration parameter calculation:

1. Function **ceil(**x**)** returns the nearest integer greater than or equal to x.
2. Function **floor(**x**)** returns the nearest integer less than or equal to x.
3. Function **max(**x1; x2;…; xn**)** returns the maximum value from the set of arguments {x1, x2,…, xn }. If the arguments xi are compound expressions composed of multiple parameters, then the values selected for each of the parameters should be the ones that maximize the overall value of xi.
4. Function **min(**x1; x2;…; xn**)** returns the minimum value from the set of arguments {x1, x2,…, xn }. If the arguments xi are compound expressions composed of multiple parameters, then the values selected for each of the parameters should be the ones that minimize the overall value of xi.
5. Function **round(**x**)** returns the integer value closest to x using asymmetric arithmetic rounding.
6. Function **if(**c; x; y**)** returns x if condition c is true, otherwise y.
7. [ ] denotes units.
8. $\mathbf{max_{M,N}(...)}$ means the maximum of all paths from node M to node N with M,N = 1, .., number of nodes and M <> N.
9. Function **or(**c1**;** c2**)** returns true if either condition c1 or condition c2 (or both) are true, otherwise returns false.
10. Function $\mathbf{min}_N(x_N)$ returns the value x of node N that represents the minimum of all nodes of a cluster.
11. Function $\mathbf{max}_N(x_N)$ returns the value x of node N that represents the maximum of all nodes of a cluster.

B.2 Bit rates

The FlexRay protocol specification defines three standard bit rates - 10 Mbit/s, 5 Mbit/s, and 2.5 Mbit/s. The configuration ranges shown in this appendix reflect the necessary parameter ranges for an implementation that supports operation at all three standard speeds.

B.3 Parameters

B.3.1 Global cluster parameters

B.3.1.1 Protocol relevant

Protocol relevant global cluster parameters are parameters used within the SDL models to describe the FlexRay protocol. They must have the same value in all nodes of a cluster.

Name	Description	Range
gColdstartAttempts	Maximum number of times a node in the cluster is permitted to attempt to start the cluster by initiating schedule synchronization.	2 - 31
gCycleCountMax	Maximum cycle counter value in a given cluster.	[7, 9 , ..., *cCycleCountMax*][a]
gdActionPointOffset	Number of macroticks the action point is offset from the beginning of a static slot.	1 - 63 MT
gdCASRxLowMax	Upper limit of the CAS acceptance window.	28 - 254 gdBit
gdDynamicSlotIdlePhase	Duration of the idle phase within a dynamic slot.	0 - 2 Minislot
gdIgnoreAfterTx	Duration that bit strobing is paused after a transmission.	0 - 15 gdBit
gdMinislot	Duration of a minislot.	2 - 63 MT
gdMinislotActionPointOffset	Number of macroticks the minislot action point is offset from the beginning of a minislot.	1 - 31 MT
gdStaticSlot	Duration of a static slot.	3 - 664 MT
gdSymbolWindow	Duration of the symbol window.	0 - 162 MT
gdSymbolWindowActionPointOffset	Number of macroticks the action point is offset from the beginning of the symbol window.	1 - 63 MT
gdTSSTransmitter	Number of bits in the Transmission Start Sequence.	1 - 15 gdBit
gdWakeupRxIdle	Number of bits used by the node to test the duration of the 'idle' or HIGH phase of a received wakeup.	8 - 59 gdBit
gdWakeupRxLow	Number of bits used by the node to test the duration of the LOW phase of a received wakeup.	8 - 59 gdBit
gdWakeupRxWindow	The size of the window, expressed in bits, used to detect wakeups.	76 - 485 gdBit
gdWakeupTxActive	Number of bits used by the node to transmit the LOW phase of a wakeup symbol and the HIGH and LOW phases of a WUDOP.	15 - 60 gdBit
gdWakeupTxIdle	Number of bits used by the node to transmit the 'idle' part of a wakeup symbol.	45 - 180 gdBit

Table B-1: Global protocol relevant parameters.

Name	Description	Range
gListenNoise	Upper limit for the startup listen timeout and wakeup listen timeout in the presence of noise. This is used as a multiplier of the node parameter *pdListenTimeout*.	2 - 16
gMacroPerCycle	Number of macroticks in a communication cycle.	8 - 16000 MT
gMaxWithoutClockCorrectionFatal	Threshold used for testing the *vClockCorrectionFailed* counter. Defines the number of consecutive even/odd cycle pairs with missing clock correction terms that will cause the protocol to transition from the *POC:normal active* or *POC:normal passive* state into the *POC:halt* state.[b]	1 - 15 even/odd cycle pairs
gMaxWithoutClockCorrectionPassive	Threshold used for testing the *vClockCorrectionFailed* counter. Defines the number of consecutive even/odd cycle pairs with missing clock correction terms that will cause the protocol to transition from the *POC:normal active* state to the *POC:normal passive* state.[b]	1 - 15 even/odd cycle pairs
gNumberOfMinislots	Number of minislots in the dynamic segment.	0 - 7988
gNumberOfStaticSlots	Number of static slots in the static segment.	2 - *cStaticSlotIDMax*
gPayloadLengthStatic	Payload length of a static frame.[c]	0 - *cPayloadLengthMax* two-byte words
gSyncFrameIDCountMax	Maximum number of distinct sync frame identifiers present in a given cluster.	2 - *cSyncFrameIDCountMax*

Table B-1: Global protocol relevant parameters.

a. must be an odd integer
b. If *gMaxWithoutClockCorrectionPassive* is set to a value greater than or equal to *gMaxWithoutClockCorrectionFatal* then the CC enters the *POC:halt* state directly (i.e., without first entering the *POC:normal passive* state).
c. All static frames in a cluster have the same payload length. For 2.5 Mbit/s the payload length is restricted by the maximum transmission duration of *adTxMax*. See section B.4.40.

B.3.1.2 Protocol related

Protocol related global cluster parameters are parameters that have a meaning in the context of the FlexRay protocol but are not used within the SDL models. These parameters are used in the configuration constraints. They must have the same value in all nodes of a cluster.

Name	Description	Range
gChannels	The channels that are used by the cluster.	[A, B, A&B]
gClockDeviationMax	Maximum frequency deviation of the time sources inside a cluster from their nominal frequencies.	0 < *gClockDeviationMax* <= *cClockDeviationMax*

Table B-2: Global protocol related parameters.

Name	Description	Range
gClusterDriftDamping	The cluster drift damping factor, based on the longest microtick *adMicrotickMax* used in the cluster. Used to compute the local cluster drift damping factor *pClusterDriftDamping*.	0 - 5 µT
gdBit	Nominal bit time.	[0.1, 0.2, 0.4] µs
gdCycle	Length of the cycle.	24 µs[a] - *cdCycleMax*
gdMacrotick	Duration of the cluster wide nominal macrotick.	1 - 6 µs
gdNIT	Duration of the Network Idle Time.	2 - 15978 MT
gdSampleClockPeriod	Sample clock period.	[0.0125, 0.025, 0.05] µs
gExternOffsetCorrection	External offset correction value applied in a cluster.	0 - 0.35 µs
gExternRateCorrection	External rate correction value applied in a cluster.	0 - 0.35 µs
gNetworkManagementVectorLength	Length of the Network Management vector in a cluster.	0 - 12 bytes

Table B-2: Global protocol related parameters.

a. See the calculation for the minimum value of *pMicroPerCycle* in section B.4.15. Maximum value is given by *cdCycleMax*. The minimum value is a theoretical minimum. Implementations may require larger minimum cycle length because of other conditions (for example performance constants given in table A-2).

B.3.2 Node parameters

B.3.2.1 Protocol relevant

Protocol relevant node parameters are parameters used within the SDL models to describe the FlexRay protocol. They may have different values in different nodes of a cluster.

Name	Description	Range
pAllowHaltDueToClock	Boolean parameter that controls the transition to the *POC:halt* state due to clock synchronization errors. If set to true, the CC is allowed to transition to *POC:halt*. If set to false, the CC will not transition to the *POC:halt* state but will enter or remain in the *POC:normal passive* state (self healing would still be possible).	Boolean
pAllowPassiveToActive	Number of consecutive even/odd cycle pairs that must have valid clock correction terms before the CC will be allowed to transition from the *POC:normal passive* state to *POC:normal active* state. If set to zero, the CC is not allowed to transition from *POC:normal passive* to *POC:normal active*.	0 - 31 even/odd cycle pairs
pChannels	Channels to which the node is connected.	[A, B, A&B]

Table B-3: Local node protocol relevant parameters.

Name	Description	Range
pClusterDriftDamping	Local cluster drift damping factor used for rate correction.	0 - 10 µT
pdAcceptedStartupRange	Expanded range of measured clock deviation allowed for startup frames during integration.	29 - 2743 µT
pDecodingCorrection	Value used by the receiver to calculate the difference between primary time reference point and secondary time reference point.	12 - 136 µT
pDelayCompensation[A], *pDelayCompensation[B]*	Value used to compensate for reception delays on the indicated channel.	4 - 211 µT
pdListenTimeout	Value for the startup listen timeout and wakeup listen timeout. Although this is a node local parameter, the real time equivalent of this value should be the same for all nodes in the cluster.	1926 - 2567692 µT
pExternalSync	Parameter indicating whether the node is externally synchronized (operating as time gateway sink in an TT-E cluster) or locally synchronized. If *pExternalSync* is set to true then *pTwoKeySlotMode* must also be set to true.	Boolean
pExternOffsetCorrection	Number of microticks added or subtracted to the NIT to carry out a host-controlled external offset correction.	0 - 28 µT
pExternRateCorrection	Number of microticks added or subtracted to the cycle to carry out a host-controlled external rate correction.	0 - 28 µT
pFallBackInternal	Parameter indicating whether a time gateway sink node will switch to local clock operation when synchronization with the time gateway source node is lost (*pFallBackInternal* = true) or will instead go to *POC:halt* (*pFallBackInternal* = false).	Boolean
pKeySlotID	ID of the key slot, i.e., the slot used to transmit the startup frame, sync frame, or designated key slot frame. If this parameter is set to zero the node does not have a key slot.	0 - *cStaticSlotIDMax*
pKeySlotOnlyEnabled	Parameter indicating whether or not the node shall enter key slot only mode following startup.	Boolean
pKeySlotUsedForStartup	Parameter indicating whether the key slot(s) are used to transmit startup frames. If *pKeySlotUsedForStartup* is set to true then *pKeySlotUsedForSync* must also be set to true. If *pTwoKeySlotMode* is set to true then both *pKeySlotUsedForSync* and *pKeySlotUsedForStartup* must also be set to true.	Boolean

Table B-3: Local node protocol relevant parameters.

Name	Description	Range
pKeySlotUsedForSync	Parameter indicating whether the key slot(s) are used to transmit sync frames. If *pKeySlotUsedForStartup* is set to true then *pKeySlotUsedForSync* must also be set to true. If *pTwoKeySlotMode* is set to true then both *pKeySlotUsedForSync* and *pKeySlotUsedForStartup* must also be set to true.	Boolean
pLatestTx	Number of the last minislot in which a frame transmission can start in the dynamic segment.	0 - 7988 Minislot
pMacroInitialOffset[A], *pMacroInitialOffset[B]*	Integer number of macroticks between the static slot boundary and the following macrotick boundary of the secondary time reference point based on the nominal macrotick duration.	2 - 68 MT
pMicroInitialOffset[A], *pMicroInitialOffset[B]*	Number of microticks between the secondary time reference point and the macrotick boundary immediately following the secondary time reference point. The parameter depends on *pDelayCompensation[Ch]* and therefore it has to be set independently for each channel.	0 - 239 µT
pMicroPerCycle	Nominal number of microticks in the communication cycle of the local node. If nodes have different microtick durations this number will differ from node to node.	960 - 1280000 µT
pOffsetCorrectionOut	Magnitude of the maximum permissible offset correction value.	15 - 16082 µT
pOffsetCorrectionStart	Start of the offset correction phase within the NIT, expressed as the number of macroticks from the start of cycle.	7 - 15999 MT
pRateCorrectionOut	Magnitude of the maximum permissible rate correction value and the maximum drift offset between two nodes operating with non-synchronized clocks for one communication cycle.	3 - 3846 µT
pSecondKeySlotID	ID of the second key slot, in which a second startup frame shall be sent when operating as a coldstart node in a TT-L or TT-E cluster. If this parameter is set to zero the node does not have a second key slot.	0 - *cStaticSlotIDMax*
pTwoKeySlotMode	Parameter indicating whether node operates as a coldstart node in a TT-E or TT-L cluster. If *pTwoKeySlotMode* is set to true then both *pKeySlotUsedForSync* and *pKeySlotUsedForStartup* must also be set to true. If *pExternalSync* is set to true then *pTwoKeySlotMode* must also be set to true.	Boolean
pWakeupChannel	Channel used by the node to send a wakeup pattern. *pWakeupChannel* must be selected from among the channels configured by *pChannels*.	[A, B]

Table B-3: Local node protocol relevant parameters.

Name	Description	Range
pWakeupPattern	Number of repetitions of the wakeup symbol that are combined to form a wakeup pattern when the node enters the *POC:wakeup send* state.	0 - 63[a]

Table B-3: Local node protocol relevant parameters.

a. A value of 0 or 1 prevents transmission of a wakeup pattern.

B.3.2.2 Protocol related

Protocol related node parameters are parameters that have a meaning in the context of the FlexRay protocol but are not used within the SDL models. They may have different values in different nodes of a cluster.

Name	Description	Range
pdMicrotick	Duration of a microtick.	[0.0125, 0.025, 0.05] µs
pNMVectorEarlyUpdate	Parameter indicating when the update of the Network Management Vector in the CHI shall take place. If *pNMVectorEarlyUpdate* is set to false, the update shall take place after the NIT. If *pNMVectorEarlyUpdate* is set to true, the update shall take place after the end of the static segment.	Boolean
pPayloadLengthDynMax	Maximum payload length for dynamic frames.[a]	0 - *cPayloadLengthMax*
pSamplesPerMicrotick	Number of samples per microtick.	[1, 2]

Table B-4: Local node protocol related parameters.

a. For bit rates of less than 10 Mbit/s the payload length is restricted by the maximum transmission duration *adTxMax*. See section B.4.41.

B.3.3 Physical layer parameters

For values of the following parameters please refer to [EPL10] and [EPLAN10].

Name	Description
dBDRx01	Time by which a positive edge is delayed in a receiving node.
dBDRx10	Time by which a negative edge is delayed in a receiving node.
dBDRxai	Idle reaction time. Time by which a transmission becomes lengthened in a receiving node (when bus is switched from active to idle). If the last actively driven bit was HIGH the idle detection in the CC is not delayed.
dBDTx01	Time by which a positive edge is delayed in a transmitting node.
dBDTx10	Time by which a negative edge is delayed in a transmitting node.
dBDTxRxai	Delay between the rising edge of the TxEN signal at the BD when TxD is still low and the RxD signal at the BD going to high.

Table B-5: Physical layer parameters.

Name	Description
dBDTxActiveMax	Maximum duration of activation of a BD's TxEN input.
dBDTxai	Propagation delay of TxEN to bus activity on a transition from bus active to bus idle, i.e., the time by which a transmission becomes lengthened in a transmitting node when bus is switched from active to idle.
dBDTxia	Propagation delay of TxEN to bus activity on a transition from bus idle to bus active, i.e., the time by which a transmission becomes shortened in a transmitting node when bus is switched from idle to active.
dBDTxDM	Absolute time difference between *dBDTxia* and *dBDTxai*.
dCCRxD01	Time by which a rising edge on the CC's RxD pin is delayed by analog effects inside the CC.
dCCRxD10	Time by which a falling edge on the CC's RxD pin is delayed by analog effects inside the CC.
dCCTxD01	Delay for the rising edge of the TxD signal between the digital domain inside the CC and the output pin of the CC.
dCCTxD10	Delay for the falling edge of the TxD signal between the digital domain inside the CC and the output pin of the CC.
dCCTxEN01	Delay for the rising edge of the TxEN signal between the digital domain inside the CC and the output pin of the CC.
dBranchRxActive-Max	Maximum duration of activity of an incoming branch of an active star.
dFrameTSSEMI-Influence$_{M,N}$	Assumed maximum shortening or lengthening of the TSS as a result of the influence of EMI effects on a TSS that is transferred from node M to node N (as a result, this term depends on the number of physical communication links, and thus the number of stars, between nodes M and N). A positive value indicates a lengthening of the TSS. This value corresponds to the [EPLAN10] parameters *dFrameTSSEMIInfluence0AS*, *dFrameTSSEMIInfluence1AS*, or *dFrameTSSEMIInfluence2AS*, depending on the number of active stars on the path between node M and node N. For the purposes of the calculations of parameter ranges, this specification assumes the minimum and maximum values from a two star system, even though [EPLAN10] places restrictions on such systems at certain bit rates.
dFrameTSS-LengthChange$_{M,N}$	Amount by which the TSS is shortened or lengthened by the network (including active stars) for a frame sent from node M to node N. This parameter does not include the stochastic effects of EMI. A positive value indicates a lengthening of the TSS. This value corresponds to the [EPLAN10] parameters *dFrameTSSLengthChange0AS*, *dFrameTSSLengthChange1AS*, or *dFrameTSSLengthChange2AS*, depending on the number of active stars on the path between node M and node N. For the purposes of the calculations of parameter ranges, this specification assumes the minimum and maximum values from a two star system, even though [EPLAN10] places restrictions on such systems at certain bit rates.[a]
dPropagation-Delay$_{M,N}$	Propagation delay from the TxD input pin of the transmitting BD (TP1_BD) of node M to the RxD output pin of the receiving BD (TP4_BD) of node N.

Table B-5: Physical layer parameters.

Name	Description
dRing	Duration of ringing on one segment of the network. If CAS/MTS or WUS symbols are transmitted, this value is used to calculate the time by which idle detection is delayed compared with a network without ringing.
$dRingRxD_{M,N}$	Maximum time including the duration of ringing and idle reaction times when ringing occurs after the transmission of a frame or WUDOP, between node M to node N. This value corresponds to the [EPLAN09] parameters $dRingRxD_0$, $dRingRxD_{AS1}$, $dRingRxD_1$, $dRingRxD_{AS2}$ or $dRingRxD_2$, depending on the number of active stars on the path between node M and node N. For the purposes of the calculations of parameter ranges, this specification assumes the maximum value from a two star system, even though [EPLAN09] places restrictions on such systems at certain bit rates.
dRxUncertainty	Time following the end of a transmission where instability may occur on RxD as a result of echoes and or ringing. During this time the RxD output may change states several times and may not reflect the actual condition of the bus.
$dStarDelay01_k$	Time by which a rising edge is delayed by star k.
$dStarDelay10_k$	Time by which a falling edge is delayed by star k.
dStarFES1Length-Change	Length change of the last bit of a frame that passes through an active star.
$nStarPath_{M,N}$	Number of stars on the signal path from any node M to a node N in a network with active stars.
dStarSymbol-EndLength-Change	Time by which the edge from low to idle after a symbol transmission is delayed by the effects of idle reaction times of the active star when a symbol passes through an active star. This time does not include the propagation delay inside the active star.
dStarTSSLength-Change	Frame TSS length change caused by an active star.
dStarTxRxai	Delay between the rising edge of the TxEN signal at the active star's CC interface when TxD is still low and the RxD signal of the active star's CC interface going to high.
$dSymbolEMI-Influence_{M,N}$	Assumed maximum shortening or lengthening of the symbol as a result of the influence of EMI effects on a symbol that is transferred from node M to node N (as a result, this term depends on the number of physical communication links, and thus the number of stars, between nodes M and N). A positive value indicates a lengthening of the symbol. This value corresponds to the [EPLAN10] parameters *dSymbolEMIInfluence0AS*, *dSymbolEMIInfluence1AS*, or *dSymbolEMIInfluence2AS*, depending on the number of active stars on the path between node M and node N. For the purposes of the calculations of parameter ranges, this specification assumes the minimum and maximum values from a two star system, even though [EPLAN10] places restrictions on such systems at certain bit rates.

Table B-5: Physical layer parameters.

Name	Description
dSymbolLength-Change$_{M,N}$	Amount by which a symbol is shortened or lengthened by the network (including active stars) for a symbol sent from node M to node N. This parameter does not include the stochastic effects of EMI. A positive value indicates a lengthening of the symbol. This value corresponds to the [EPLAN10] parameters *dSymbolLengthChange0AS*, *dSymbolLengthChange1AS*, or *dSymbolLengthChange2AS*, depending on the number of active stars on the path between node M and node N. For the purposes of the calculations of parameter ranges, this specification assumes the minimum and maximum values from a two star system, even though [EPLAN10] places restrictions on such systems at certain bit rates.
dWU$_{0Detect}$	Acceptance timeout for detection of a LOW phase in a wakeup pattern. The maximum value of this parameter represents a duration that will be accepted as a LOW phase by all BDs.
dWU$_{IdleDetect}$	Acceptance timeout for detection of an Idle phase in a wakeup pattern. The maximum value of this parameter represents a duration that will be accepted as an Idle phase by all BDs.
dWU$_{Timeout}$	Acceptance timeout for wakeup pattern recognition. The minimum value of this parameter represents a pattern duration that would be accepted as a wakeup by all BDs.

Table B-5: Physical layer parameters.

a. The frame decoding mechanism, a portion of which is described in Figure 3-28, relies on the assumption that the effects of the TSS length change and the TSS EMI influences, combined with the appropriate bit strobing and quantization effects, can increase the receiver's perspective of the length of a TSS by at most two bit times. The maximum values for *dFrameTSSLengthChange* and *dFrameTSSEMIInfluence* given in [EPL10] and [EPLAN10] satisfy this assumption.

B.3.4 Auxiliary parameters

The following parameters are only introduced for configuration constraints.

Name	Description
aAssumedPrecision	Assumed precision of the application network.
aBestCasePrecision	Upper bound for the clock deviation between two nodes (precision) assuming no faults are present in the cluster.
adActionPointDifference	Amount by which the static slot action point offset is greater than the minislot action point offset (zero if static slot action point is smaller than minislot action point).
adBitMax	Maximum bit time taking into account the allowable clock deviation of each node.
adBitMin	Minimum bit time taking into account the allowable clock deviation of each node.
adDTSLow	Duration of the low phase of the Dynamic Trailing Sequence.
adInitializationErrorMax	The maximum initialization error that must be tolerated by an integrating node.

Table B-6: Auxiliary parameters for configuration constraints.

Name	Description
adInternalRxDelay	Additional implementation dependent delay on the receive path of the decoder up to the strobe point, i. e. an additional synchronization unit in front of the majority voting mechanism. It is in the responsibility of the semiconductor manufacturer to specify this implementation dependent value for its devices.
adLineDelay[Ch]$_{M,N}$	The contribution of the propagation delay attributed to the path lengths and the specific line delays T'_0 of the various segments of the communication path between node M and node N (see [EPL10]). This value also includes delays attributed to the circuit board traces between the CC and the BD in both the transmitter and the receiver.
adMaxIdleDetectionDelayAfterHIGH	Maximum time by which idle detection is delayed when ringing occurs after the transmission of a frame or WUDOP.
adMicrotickDistError	Maximum time difference in a node arising from the integral (i.e., non-fractional) distribution of microticks across different macroticks vs. an ideal distribution that allowed fractional microticks. The value is based on the local microtick *pdMicrotick*.
adMicrotickMax	Maximum microtick length of all microticks configured within a cluster.
adMicrotickMaxDistError	Maximum time difference in a cluster arising from the integral (i.e., non-fractional) distribution of microticks across different macroticks vs. an ideal distribution that allowed fractional microticks. The value is based on the largest microtick in the cluster, *adMicrotickMax*.
adOffsetCorrection	The duration in macroticks of the offset correction phase of the NIT.
adPropagationDelayMax	Maximum propagation delay of a cluster.
adPropagationDelayMin	Minimum propagation delay of a cluster.
adRemOffsetCalculation	Time after the beginning of the NIT necessary to ensure completion of the offset correction calculation.
adRemRateCalculation	Time after the beginning of the NIT necessary to ensure completion of the rate correction calculation.
adSymbolWindowGuardInterval	Required period of inactivity to ensure that symbols transmitted in the symbol window are perceived by all receivers as beginning in the symbol window.
adTxDyn	Upper bound on the duration of a dynamic frame transmission.
adTxMax	Maximum transmission duration of a CC (expressed in µs).
adTxStat	Upper bound on the duration of a static frame transmission.
aFrameLength	Frame length in bits of a frame including transmission start sequence, frame start sequence and frame end sequence but without idle detection time.
aFrameLengthDynamic	Frame length in bits of a dynamic frame (see *aFrameLength*).
aFrameLengthStatic	Frame length in bits of a static frame (see *aFrameLength*)

Table B-6: Auxiliary parameters for configuration constraints.

Name	Description
aMicroPerMacroNom	The nominal number of microticks per macrotick for a node. Note that this number need not be an integer. The allowable range for this auxiliary variable is bounded by *cMicroPerMacroNomMin* and *cMicroPerMacroNomMax*.
aMinislotPerDynamicFrame	Number of minislots needed to transmit a frame in the dynamic segment.
aMixedTopologyError	A term that expresses in microseconds the effect that the difference in propagation delay between the time source and time sink clusters has on the precision of the time sink cluster.
aNegativeOffsetCorrectionMax	The maximum amount of time by which the offset correction phase of the NIT might need to be shortened to account for negative offset corrections.
$anRingPath_{M,N}$	Number of segments between node M and node N where ringing occurs.
aOffsetCorrectionMax	Cluster global magnitude of the maximum necessary offset correction value.
aPayloadLength	Payload length in two-byte words.
aPayloadLengthDynamic	Payload length in two-byte words of a dynamic frame.
aPositiveOffsetCorrectionMax	The maximum amount of time by which the offset correction phase of the NIT might need to be lengthened to account for positive offset corrections.
aSinkPrecision	An upper bound for the clock deviation between two nodes in the time sink cluster in a TT-E system. Different definitions of this parameter exist depending on the nature of the time source cluster.
aWorstCasePrecision	Upper bound for the clock deviation between two nodes (precision) when a limited number of Byzantine faults are present in the clock synchronization of the cluster (refer to [Ung09] for details).

Table B-6: Auxiliary parameters for configuration constraints.

B.4 Calculation of configuration parameters for nodes in a TT-D cluster

This section describes how the configuration parameters are calculated for nodes operating in TT-D cluster. Unless specified differently in B.5, B.6, or B.7 these calculations also apply for nodes operating in TT-L or TT-E clusters.

B.4.1 gClockDeviationMax

Clock sources for communication controllers in a cluster deviate from each other. The parameter *gClockDeviationMax* defines the maximum clock frequency deviation of any node in the cluster from the node's nominal clock frequency, defined as a ratio of the absolute value of the maximum frequency deviation to the nominal clock frequency, i.e., the actual clock frequency will be in the range

[1] (1 - *gClockDeviationMax*) * $f_{nominal}$ <= f_{actual} <= (1 + *gClockDeviationMax*) * $f_{nominal}$

where $f_{nominal}$ is the node's nominal clock frequency and f_{actual} is the node's actual clock frequency. The definition of certain startup mechanisms and the calculation of the ranges of the FlexRay configuration parameters makes it necessary to limit the maximum allowed clock deviation in a cluster to the protocol constant *cClockDeviationMax*:

Constraint 1:

0 < *gClockDeviationMax* <= *cClockDeviationMax*

Note that physical layer effects such as asymmetric propagation delays of rising and falling edges may put additional constraints on the maximum allowable clock deviation for certain physical layer topologies. Refer to [EPLAN10] for additional details.

The calculation of the parameter ranges in this appendix is based on an assumption that *gClockDeviationMax* = *cClockDeviationMax* but for simplicity *gClockDeviationMax* is not included in the tables showing the calculation of the parameter ranges.

B.4.2 Attainable precision

Various error terms influence the attainable precision of the FlexRay clock synchronization algorithm (see [Ung09]). In order to choose proper configuration parameters it is necessary to know the attainable precision of the application network. In order to simplify the equations the following calculations assume that the configurations for propagation delay compensation are configured to a value of identical time in all nodes (see section B.4.25 for details).

B.4.2.1 Propagation Delay

B.4.2.1.1 adInternalRxDelay

An implementation of the protocol may, for practical reasons, introduce an additional delay between the RxD pin of the communication controller and the circuits responsible for majority voting and bit strobing[187] (see section 3.2.4).

The allowed range of *adInternalRxDelay* for an implementation is

[2] *cdInternalRxDelayMin* <= *adInternalRxDelay* <= *cdInternalRxDelayMax*.

The parameter *adInternalRxDelay* is implementation dependent and, like other implementation dependent behavior, must be specified in the documentation of an implementation.

B.4.2.1.2 adPropagationDelayMax

A parameter for the maximum propagation delay *adPropagationDelayMax*[µs] of the cluster is introduced with

Constraint 2:

$$adPropagationDelayMax[Ch][\mu s] = \mathbf{max_{M,N}(}\ adLineDelay[Ch]_{M,N}[\mu s] + \mathbf{max(}\ dCCTxD01[Ch]_{M}[\mu s] + dBDTx01[Ch]_{M}[\mu s] + \sum_{k \in \{nStarPath[Ch]_{MN}\}} dStarDelay01_{k}\ [\mu s] + dBDRx01[Ch]_{N}[\mu s] + dCCRxD01[Ch]_{N}[\mu s]\ ;$$

[187] This could be, for example, an additional sample delay for evaluation of *zVotedVal* and setting *zSampleCounter* or an additional synchronization register prior to the voting registers (to avoid metastability effects).

$dCCTxD10[Ch]_M$[µs] + $dBDTx10[Ch]_M$[µs] +

$$\sum_{k \in \{nStarPath[Ch]_{MN}\}} dStarDelay10_k$$ [µs] + $dBDRx10[Ch]_N$[µs] + $dCCRxD10[Ch]_N$[µs] **)** +

(1 samples + $adInternalRxDelay_N$[samples] + *cVotingDelay*[samples] +

(*cStrobeOffset*[samples] - 1 samples)) *

gdSampleClockPeriod[µs/samples] / (1 - *gClockDeviationMax*) **)**

[3] *adPropagationDelayMax*[µs] = **max(** *adPropagationDelayMax[A]*[µs]**;** *adPropagationDelayMax[B]*[µs] **)**

[4] *adPropagationDelayMin*[µs] = **min(** *adPropagationDelayMin[A]*[µs]**;** *adPropagationDelayMin[B]*[µs] **)**

- $dCCTxD01[Ch]_M$ and $dBDTx01[Ch]_M$ are the contributions to the propagation delay of a rising edge caused by analog effects of the CC and BD of the transmitting node *M* on channel *Ch*.
- $dCCTxD10[Ch]_M$ and $dBDTx10[Ch]_M$ are the contributions to the propagation delay of a falling edge caused by analog effects of the CC and BD of the transmitting node *M* on channel *Ch*.
- $dBDRx01[Ch]_N$ and $dCCRxD01[Ch]_N$ are the contributions to the propagation delay of a rising edge caused by analog effects of the BD and CC of the receiving node *N* on channel *Ch*.
- $dBDRx10[Ch]_N$ and $dCCRxD10[Ch]_N$ are the contributions to the propagation delay of a falling edge caused by analog effects of the BD and CC of the receiving node *N* on channel *Ch*
- $dStarDelay01_k$ is the propagation delay of a rising edge and $dStarDelay10_k$ is the propagation delay of a falling edge of star *k*.
- $adLineDelay[Ch]_{M,N}$ is the contribution of the propagation delay attributed to the path lengths and the specific line delays T'_0 of the various segments of the communication path between node *M* and node *N* (see [EPL10]).[188]
- $nStarPath[Ch]_{M,N}$ is the number of active stars between node *M* and *N* on Channel *Ch*.
- 1 sample is the delay between the edge occurring on the bus and the sampling edge.
- $adInternalRxDelay_N$[samples] is the implementation dependent delay on the receive path of the decoder up to the strobe point of node *N*.
- *cVotingDelay* ist the delay introduced by the majority voting.
- *cStrobeOffset* - 1 sample is the delay introduced by the bit strobing process (BITSTRB).

For the purpose of deriving ranges for parameters within this appendix it is assumed that the portion of the propagation delay caused by the physical layer and the analog effects of the FlexRay CC's involved in communication is bounded by *cPropagationDelayMax*; this is true for all clusters adhering to [EPL10]. This leads to the following equation:

[188] Any time delay introduced by the circuit board traces between the TxD output of the CC and the TxD input of the BD (on the transmitter) and the RxD output of the BD and the RxD input of the CC (on the receiver) are assumed to be included in the $adLineDelay[Ch]_{M,N}$ parameter. The contributions of these terms are expected to be small (on the order of 1-2 ns), and certainly smaller than the value implied for this contribution by the difference between *cPropagationDelayMax* and $dPropagationDelay_{M,N}$.

[5] *adPropagationDelayMax*^{Max}[µs] = **max$_N$(** *cPropagationDelayMax*[µs] + (1 samples + *adInternalRxDelay*$_N$[samples] + *cVotingDelay*[samples] + (*cStrobeOffset*[samples] - 1 samples)) * *gdSampleClockPeriod*[µs/samples] / (1 - *gClockDeviationMax*) **)**

Bit Rate [Mbit/s]	2.5	5	10
gdSampleClockPeriod[µs]	0.050	0.025	0.0125
max$_N$(*adInternalRxDelay*$_N$^{Max}[samples] **)**	4		
adPropagationDelayMax^{Max}[µs]	**3.051**	**2.775**	**2.638**

Table B-7: Calculations for *adPropagationDelayMax*^{Max}.

The value of *adPropagationDelayMax* should be estimated for the given network topology using the estimated propagation delay of transmitter, line, star, receiver, and the communication controller internal delay.

To allow calcuation of the minimum value of *adPropagationDelayMax* it is assumed that physical layer effects of Constraint 2 are zero. With this one gets

[6] *adPropagationDelayMax*^{Min}[µs] = **max$_N$(** (1 samples + *adInternalRxDelay*$_N$[samples] + *cVotingDelay*[samples] + *cStrobeOffset*[samples] - 1 samples) * *gdSampleClockPeriod*[µs/samples] / (1 - *gClockDeviationMax*) **)**

Bit Rate [Mbit/s]	2.5	5	10
gdSampleClockPeriod[µs]	0.050	0.025	0.0125
max$_N$(*adInternalRxDelay*$_N$^{Min}[samples] **)**	1		
adPropagationDelayMax^{Min}[µs]	**0.401**	**0.200**	**0.100**

Table B-8: Calculations for *adPropagationDelayMax*^{Min}.

B.4.2.1.3 adPropagationDelayMin

The formula for *adPropagationDelayMin* takes following effects into account:

- physical layer effects as listed for *adPropagationDelayMax*,
- the effects of the decoding unit, i.e., the majority voting delay and the strobing delay,
- the minimum of the internal delays caused by the implementations.

Without taking the physical layer influences into account[189] one gets

[189] Obviously it is not possible for the physical layer to introduce no propagation delay, but an assumption of zero propagation delay is conservative and would result in parameter ranges that would accommodate any practical minimum propagation delay.

Constraint 3:

adPropagationDelayMin[µs] >=

min$_N$(*adInternalRxDelay*$_N$[samples] + *cVotingDelay*[samples] +

(*cStrobeOffset*[samples] - 1 samples) **)** * *gdSampleClockPeriod*[µs/samples] /

(1 + *gClockDeviationMax*)

Taking the physical layer effects into account one gets

Constraint 4:

adPropagationDelayMin[Ch][µs] = **min$_{M,N}$(** *adLineDelay[Ch]*$_{M,N}$[µs] +

min(*dCCTxD01[Ch]*$_M$[µs] + *dBDTx01[Ch]*$_M$[µs] +

$$\sum_{k \in \{nStarPath[Ch]_{MN}\}} dStarDelay01_k$$ [µs] + *dBDRx01[Ch]*$_N$[µs] + *dCCRxD01[Ch]*$_N$[µs] **;**

dCCTxD10[Ch]$_M$[µs] + *dBDTx10[Ch]*$_M$[µs] +

$$\sum_{k \in \{nStarPath[Ch]_{MN}\}} dStarDelay10_k$$ [µs] + *dBDRx10[Ch]*$_N$[µs] + *dCCRxD10[Ch]*$_N$[µs] **)** +

(*adInternalRxDelay*$_N$[samples] + *cVotingDelay*[samples] +

(*cStrobeOffset*[samples] - 1 samples)) *

gdSampleClockPeriod[µs/samples] / (1 + *gClockDeviationMax*) **)**

With Constraint 3 the minimum can be calculated to be

Bit Rate [Mbit/s]	**2.5**	**5**	**10**
gdSampleClockPeriod[µs]	0.050	0.025	0.0125
min$_N$(*adInternalRxDelay*$_N^{Min}$[samples] **)**	1		
adPropagationDelayMin^{Min}[µs]	**0.349**	**0.175**	**0.087**

Table B-9: Calculations for *adPropagationDelayMin*^{Min}.

To allow calculation of the maximum value of *adPropagationDelayMin* it is assumed that the physical layer effects of Constraint 4 are bounded by the upper limit of *cPropagationDelayMax*. With this one gets:

[7] *adPropagationDelayMin*^{Max}[µs] = **min$_N$(** *cPropagationDelayMax*[µs] + (*adInternalRxDelay*$_N$[samples] + *cVotingDelay*[samples] + *cStrobeOffset*[samples] - 1 samples) * *gdSampleClockPeriod*[µs/samples] / (1 + *gClockDeviationMax*) **)**

Bit Rate [Mbit/s]	2.5	5	10
gdSampleClockPeriod[µs]	0.050	0.025	0.0125
min$_N$(*adInternalRxDelay*$_N^{Max}$[samples] **)**	4		
adPropagationDelayMin^{Max}[µs]	**2.999**	**2.750**	**2.625**

Table B-10: Calculations for *adPropagationDelayMin*^{Max}.

B.4.2.2 Microtick Distribution Error

The macrotick generation process generates macrotick events, which are naturally restricted to a discrete grid determined by the local microtick clock. Compared to an ideal macrotick generation process, which would be able to generate events at any point in time, the restriction to the microtick grid introduces a discretization error. The macrotick generation process is designed to prevent the accumulation of this discretization error; it is assured that the duration of any number of consecutive, discrete macroticks only deviates from the duration of the corresponding ideal, non-discrete macroticks by at most the duration of a single microtick. This single microtick must be taken into consideration when converting an amount of real time into a number of discrete local macroticks of a node.

The auxiliary variable *adMicrotickDistError* is introduced to capture the effect of the discretization of the macrotick clock in a given node:

[8] *adMicrotickDistError*[µs] = *pdMicrotick*[µs] / (1 - *gClockDeviationMax*)

***pdMicrotick*[µs]**	**0.050**	**0.025**	**0.0125**
adMicrotickDistError[µs]	**0.050075**	**0.025038**	**0.012519**

Table B-11: Calculations for *adMicrotickDistError*.

The values for *adMicrotickDistError* for various microtick durations are calculated in table B-11.

The auxiliary variable *adMicrotickMaxDistError* is introduced to capture the effect of the discretization of the macrotick clocks in a cluster, and is therefore based on the longest microtick in a cluster:

[9] *adMicrotickMaxDistError*[µs] = *adMicrotickMax*[µs] / (1 - *gClockDeviationMax*)

***adMicrotickMax*[µs]**	**0.050**	**0.025**	**0.0125**
adMicrotickMaxDistError[µs]	**0.050075**	**0.025038**	**0.012519**

Table B-12: Calculations for *adMicrotickMaxDistError*.

The values for *adMicrotickMaxDistError* for various microtick durations are calculated in table B-12. These values are used in the calculations of parameter ranges but are not specifically included in the tables.

B.4.2.3 Worst-case precision

First of all, a parameter defining the maximum microtick length of a cluster is introduced:

[10] *adMicrotickMax*[µs] = **max(** { x | x = *pdMicrotick* of each node } **)**

The worst-case error before the clock correction[190] is given by

[11] *aWorstCasePrecision*[µs] = (38 µT + 20 * *gClusterDriftDamping*[µT]) * *adMicrotickMax*[µs/µT] /
(1 - *gClockDeviationMax*) + *adMicrotickMaxDistError*[µs] +
2 * (*adPropagationDelayMax*[µs] - *adPropagationDelayMin*[µs])

It is important to note that the attainable precision directly depends on network topology (*adPropagation-DelayMax*) and the maximum microtick used in the cluster (*adMicrotickMax*).

Examinations within the FlexRay consortium have shown that a value of *gClusterDriftDamping* = 5 µT is enough to prevent cluster drifts. Since cluster precision becomes worse as *gClusterDriftDamping* increases, *gClusterDriftDamping* is limited to 5 µT.

Bit Rate [Mbit/s]	**2.5**	**5**		**10**	
***adMicrotickMax*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
gClusterDriftDamping^{Max}[µT]	5				
adPropagationDelayMax^{Max}[µs]	3.051	2.775		2.638	
adPropagationDelayMin^{Min}[µs]	0.349	0.175		0.087	
aWorstCasePrecision^{Max}[µs]	**12.364**	**12.160**	**8.680**	**8.582**	**6.842**

Table B-13: Calculation of the worst-case precision.

B.4.2.4 Best-case precision

The best-case precision can be calculated using a simplified version of equation [11] which does not take Byzantine errors into account.

[12] *aBestCasePrecision*[µs] = (13 µT + 6 * *gClusterDriftDamping*[µT]) * *adMicrotickMax*[µs/µT] /
(1 - *gClockDeviationMax*) + *adMicrotickMaxDistError*[µs] +
adPropagationDelayMax[µs] - *adPropagationDelayMin*[µs]

[190] Please note that in case the correction cannot be performed at the end of a double cycle, e.g. because no sync frames could be received due to a transitory disturbance on the network, the precision of the cluster can get worse than the value indicated in equation [11]. This should be taken into consideration if nodes of a cluster are allowed to remain in *POC:normal active* while no clock correction values could be derived, i.e. when *gMaxWithoutClockCorrectionPassive* is configured to be larger than 1. While the additional worsening of the precision will generally be rather small, an estimate depends heavily on the underlying error assumptions and is beyond the scope of this document.

It is not possible to reach a precision better than calculated in equation [12].

Bit Rate [Mbit/s]	2.5	5		10	
***adMicrotickMax*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
gClusterDriftDamping^{Min}[µT]	0				
min(*adPropagationDelayMax*[µs] - *adPropagationDelayMin*[µs] **)**	0				
aBestCasePrecision^{Min}[µs]	**0.701**	**0.701**	**0.351**	**0.351**	**0.175**

Table B-14: Calculation of the best-case precision.

B.4.2.5 Assumed precision

An assumed precision of the cluster, *aAssumedPrecision* is introduced. The *aAssumedPrecision* parameter is necessary to derive additional constraints on other timing parameters. This parameter is given by

Constraint 5:

aAssumedPrecision[µs] >= *aBestCasePrecision*[µs]

For purposes of configuration of the cluster, it is suggested that the assumed precision be set equal to the worst case precision:

[13] *aAssumedPrecision*[µs] = *aWorstCasePrecision*[µs]

Parameter ranges in this specification are calculated assuming that the maximum and minimum value of *aAssumedPrecision* are *aWorstCasePrecision*^{Max} and *aBestCasePrecision*^{Min}, respectively.

B.4.3 Ringing

Due to physical layer effects, a node may experience a period of instability on the physical layer after the end of a transmission. During this period the bus level may exceed the thresholds for detecting active low and active high and may switch multiple times between the active low and active high state, ending on either of these states. This is called ringing.

The duration of ringing depends mainly on the topology, especially on the number of active stars. For details refer to section 2.17 in [EPLAN10].

From the perspective of a communication controller ringing can happen after the end of a frame or symbol transmission and would show up as one or multiple transitions between LOW and HIGH on the RxD input.

Ringing has various effects - some of them are listed below:

- After the reception of a FES high bit, ringing delays the start of the idle phase.
- After the reception of a DTS high bit, ringing delays the start of the idle phase. Ringing can trigger additional *potential idle start* signals sent to the MAC process, which has an influence in case of a dynamic frame decoding error.
- After the reception of a symbol, ringing delays the start of the idle phase.
- After the transmission of a WUS low phase, ringing may appear to the transmitting node as if a collision with another node's WUS low phase had happened.

When ringing occurs after a transmission that ends on a high bit (which is the case for frames and WUDOPs), from the perspective of the physical layer the start of idle detection is delayed by the duration of

ringing itself plus the idle reaction times within the network. This delay is specified by the parameter *dRingRxD*$_{M,N}$ as described in section 2.17 in [EPLAN10].

In cases where no ringing occurs the reception of the last transmitted high bit is counted as idle by the idle detection mechanism of the receiving CCs - for them the bus needs to be idle for only (*cChannelIdleDelimiter* - 1) gdBit after the last high bit was transmitted (which is of course taken into account by the configuration constraints). If ringing occurs, however, the last high bit that was received before ringing cannot be taken into account for idle detection, and therefore the receiving CCs need to see HIGH for *cChannelIdleDelimiter* gdBit after the end of the idle reaction time.

The overall time by which idle detection is delayed inside the receiving CCs is described by the following equation (and will be zero if no ringing occurs):

[14] *adMaxIdleDetectionDelayAfterHIGH*[µs] = **max$_{M,N}$(if(** *dRingRxD*$_{M,N}$[µs] > 0**;**
dRingRxD$_{M,N}$[µs] + *adBitMax*[µs]**;** 0 **))**

When ringing occurs after a transmission that ends on a low bit (which is the case for CAS/MTS and WUS symbols) idle detection is only delayed beyond the time already considered by the symbol length change parameter by the actual duration of the ringing.

The duration of ringing is specified by the parameter *dRing* as described in section 2.17 in [EPLAN10].

In principle, ringing can occur on every segment of the cluster except the link between two active stars. The auxiliary variable *anRingPath*$_{M,N}$ is introduced to define the number of segments that could ring between node *M* and node *N* in a given cluster:

[15] 0 <= *anRingPath*$_{M,N}$ <= 2

The maximum overall duration of ringing in a given cluster is then **max$_{M,N}$(** *anRingPath*$_{M,N}$ * *dRing*[µs] **)**.

B.4.4 Definition of microtick, macrotick, and bit time

The parameter *pdMicrotick* is usually not a direct configuration parameter of an implementation, instead being derived from the sample clock rate and the selected value for *pSamplesPerMicrotick*. It is introduced here because it is used to derive various parameter constraints.

The node-specific microtick length (*pdMicrotick*) is application dependent and may be different for each node. The bit length (*gdBit*) must be identical for all nodes of the cluster. Both parameters are defined in multiples of the sample clock period.

[16] *pdMicrotick*[µs] = *pSamplesPerMicrotick* * *gdSampleClockPeriod*[µs]

[17] *gdBit*[µs] = *cSamplesPerBit* * *gdSampleClockPeriod*[µs]

Due to clock tolerances the duration of a bit may vary from the ideal bit timing. The maximum and minimum possible bit duration depends on the clock tolerance and can be calculated as follows:

[18] *adBitMax*[µs] = *gdBit*[µs] / (1 - *gClockDeviationMax*)

[19] *adBitMin*[µs] = *gdBit*[µs] / (1 + *gClockDeviationMax*)

Bit Rate [Mbit/s]	2.5	5	10
gdBit[µs]	0.4	0.2	0.1
adBitMax[µs]	0.4006	0.2003	0.10015

Table B-15: Calculations for *adBitMax* and *adBitMin*.

Bit Rate [Mbit/s]	2.5	5	10
adBitMin[µs]	0.3994	0.1997	0.09985

Table B-15: Calculations for *adBitMax* and *adBitMin*.

Table B-16 shows the possible microtick lengths depending on *pSamplesPerMicrotick* and *gdSampleClock-Period*. An implementation shall support all of the combinations shown in this table.

Bit Rate [Mbit/s]	2.5	5		10	
***gdSampleClockPeriod*[µs]**	**0.050**	**0.025**		**0.0125**	
pSamplesPerMicrotick	1	2	1	2	1
pdMicrotick[µs]	0.050	0.050	0.025	0.025	0.0125

Table B-16: *pdMicrotick* depending on *pSamplesPerMicrotick* and *gdSampleClockPeriod*.

Table B-17 shows representative values of the nominal macrotick length *gdMacrotick* that can be achieved with various microtick durations while keeping both *aMicroPerMacroNom* and *gdMacrotick* within their allowable ranges. The actual macrotick duration used in a system need not be selected from those listed in the table, and need not even use integral values for *aMicroPerMacroNom*, but *aMicroPerMacroNom* must be in the range of 40 to 240 microticks per macrotick, *gdMacrotick* must be in the range of 1 to 6 micro-seconds, and Constraint 7 must be fulfilled.

aMicroPer-MacroNom	***pdMicrotick*[µs]**		
	0.0125	0.025	0.050
40	-	1	2
60	-	1.5	3
80	1	2	4
120	1.5	3	6
240	3	6	-

Table B-17: Examples of *gdMacrotick* as a function of *aMicroPerMacroNom* and *pdMicrotick*.

The desired nominal macrotick length *gdMacrotick* in µs can be chosen to be less than, greater than, or equal to the assumed precision. In all cases the macrotick length must fulfill the following constraint:

Constraint 6:

cdMinMTNom[µs] <= *gdMacrotick*[µs] <= *cdMaxMTNom*[µs]

Depending on the desired macrotick length the following additional constraint must be met.

Constraint 7:

gdMacrotick[µs] >= *cMicroPerMacroNomMin*[µT] * *pdMicrotick*[µs/µT]

A parameter describing the nominal number of microticks in a macrotick is given by

[20] *aMicroPerMacroNom*[µT/MT] = *gdMacrotick*[µs/MT] / *pdMicrotick*[µs/µT]

For *gdMacrotick* there is an additional constraint because of the startup procedure and *pMicroInitialOffset*:

Constraint 8:

gdMacrotick[µs] <= ((*aFrameLengthStatic*[gdBit] - *cdFES*[gdBit]) * *gdBit*[µs/gdBit] -

gdSampleClockPeriod[µs] + *adPropagationDelayMin*[µs]) *

((1 - *gClockDeviationMax*) / (1 + *gClockDeviationMax*))

Constraint 8 is necessary to ensure that events occur in the order required by the mechanism described in Figure 8-11, in particular, that at least one macrotick length must pass between the primary time reference point and the *valid odd startup frame on A* signal in order for the *continue integration on A* signal to properly start the macrotick generation. Note, however, that fulfillment of this constraint is automatic because of the maximum macrotick length allowed by the protocol and the allowed values of minimum frame length, bit duration, and the other parameters in the constraint.

B.4.5 adInitializationErrorMax

The maximum initialization error that must be tolerated by an integrating node depends primarily on the assumed precision and the propagation delay. Consider the following assumptions:

- The clocks of two nodes may have an offset of up to the assumed precision.
- An external offset correction might need to be applied.
- The received frames might be received with the maximum propagation delay of which only the minimal possible propagation delay is compensated by the local delay compensation.

Constraint 9:

adInitializationErrorMax[µs] >= *aAssumedPrecision*[µs] + *gExternOffsetCorrection*[µs] +

adPropagationDelayMax[µs] - *adPropagationDelayMin*[µs]

In general, *adInitializationErrorMax* should be chosen using the equality constraint. Under certain exceptional circumstances, however, this choice of parameter could result in a slight delay of startup. The system designer may choose to reduce this probability by selecting a somewhat larger value for *adInitializationErrorMax*, and thus the previous constraint is expressed as an inequality. Note that increasing the value of *adInitializationErrorMax* may have a significant impact on the required action point offset, and therefore the required static slot size.

Bit Rate [Mbit/s]	**2.5**	**5**		**10**	
***adMicrotickMax*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
$aAssumedPrecision^{Min}$[µs] = $aBestCasePrecision^{Min}$[µs]	0.701	0.701	0.351	0.351	0.175
$gExternOffsetCorrection^{Min}$[µs]	0				
adPropagationDelayMax[µs] - *adPropagationDelayMin*[µs]	0				
$adInitializationErrorMax^{Min}$[µs]	**0.701**	**0.701**	**0.351**	**0.351**	**0.175**
$aAssumedPrecision^{Max}$[µs] = $aWorstCasePrecision^{Max}$[µs]	12.364	12.160	8.680	8.582	6.842
$gExternOffsetCorrection^{Max}$[µs]	0.35				

Table B-18: Calculations for *adInitializationErrorMax*.

Bit Rate [Mbit/s]	2.5	5		10	
***adMicrotickMax*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
adPropagationDelayMin^{Min}[µs]	0.349	0.175		0.087	
adPropagationDelayMax^{Max}[µs]	3.051	2.775		2.638	
adInitializationErrorMax^{Max}[µs]	**15.416**	**15.110**	**11.630**	**11.483**	**9.743**

Table B-18: Calculations for *adInitializationErrorMax*.

B.4.6 pdAcceptedStartupRange

Consider the following assumptions:

- During integration a clock synchronization error greater than the assumed precision may occur and be acceptable.

Constraint 10:

pdAcceptedStartupRange[µT] >= **ceil(** (*aAssumedPrecision*[µs] + *adInitializationErrorMax*[µs]) / (*pdMicrotick*[µs/µT] / (1 + *gClockDeviationMax*)) **)**

Bit Rate [Mbit/s]	2.5	5		10	
***adMicrotickMax*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
aAssumedPrecision^{Min}[µs] = *aBestCasePrecision*^{Min}[µs]	0.701	0.701	0.351	0.351	0.175
adInitializationErrorMax^{Min}[µs]	0.701	0.701	0.351	0.351	0.175
pdMicrotick^{Max}[µs]	0.050	0.050	0.025	0.025	0.0125
pdAcceptedStartupRange^{Min}[µT]	**29**	**29**	**29**	**29**	**29**
aAssumedPrecision^{Max}[µs] = *aWorstCasePrecision*^{Max}[µs]	12.364	12.160	8.680	8.582	6.842
adInitializationErrorMax^{Max}[µs]	15.416	15.110	11.630	11.483	9.743
pdMicrotick^{Min}[µs]	0.050	0.025		0.0125	
pdAcceptedStartupRange^{Max}[µT]	**557**	**1093**	**814**	**1608**	**1329**

Table B-19: Calculations for *pdAcceptedStartupRange*.

The lower bound of *pdAcceptedStartupRange* is given by the calculation in a TT-D cluster and the upper bound by the calculation in a TT-E cluster (see section B.7.13, table B-49). As a result, the parameter *pdAcceptedStartupRange* must be configurable over a range of 29 to 2743 µT.

B.4.7 pClusterDriftDamping

Consider the following assumptions:

- The drift damping factor *gClusterDriftDamping* is defined in multiples of the longest microtick *adMicrotickMax* within the cluster:

  gClusterDriftDamping[µT] = n * 1 µT with n = 0, 1, 2, ..., 5.

- The maximum microtick of a cluster *adMicrotickMax* is a multiple m of each node's local microtick *pdMicrotick*.

  adMicrotickMax = m * *pdMicrotick* with m = 1, 2 (see *pSamplesPerMicrotick*).
- The local drift damping *pClusterDriftDamping* is calculated by

Constraint 11:

pClusterDriftDamping[µT] <= *gClusterDriftDamping*[µT] * (*adMicrotickMax*[µs/µT] / *pdMicrotick*[µs/µT])

Constraint 11 should be treated as a recommendation. In practice it is expected that the drift damping factor is chosen such that

$$pClusterDriftDamping \approx gClusterDriftDamping$$

The upper limit of *pClusterDriftDamping* is given by *adMicrotickMax*/*pdMicrotick* = 2. Therefore, $pClusterDriftDamping^{Max}$ = 2 * 5 µT = 10 µT, and thus the parameter *pClusterDriftDamping* must be configurable over a range of 0 to 10 µT.

B.4.8 gdActionPointOffset

Consider the following assumptions:

- The action point offset should be greater than the assumed precision.
- A minimum propagation delay of the network as seen by the local node is given by *adPropagationDelayMin*[µs].

Constraint 12:

gdActionPointOffset[MT] >= **ceil(** (*aAssumedPrecision*[µs] - *adPropagationDelayMin*[µs]) / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **)**

In order to prevent the possibility of the creation of cliques[191] during startup an additional safety margin must be added. In this case Constraint 13 replaces Constraint 12.

Constraint 13:

gdActionPointOffset[MT] >= **ceil(** (2 * *aAssumedPrecision*[µs] - *adPropagationDelayMin*[µs] + 2 * *adInitializationErrorMax*[µs]) / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **)**

Bit Rate [Mbit/s]	**2.5**	**5**		**10**	
***adMicrotickMax*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
$aAssumedPrecision^{Max}$[µs] = $aWorstCasePrecision^{Max}$[µs]	12.364	12.160	8.680	8.582	6.842
$adPropagationDelayMin^{Min}$[µs]	0.349	0.175		0.087	
$adInitializationErrorMax^{Max}$[µs]	15.416	15.110	11.630	11.483	9.743

Table B-20: Calculations for *gdActionPointOffset*.

[191] Clique formation may be possible if more than two coldstart nodes are configured in the cluster under specific error scenarios.

Bit Rate [Mbit/s]	2.5	5		10	
***adMicrotickMax*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
gdMacrotick^{Min}[µs]	2	2	1	1	1
gdActionPointOffset^{Max}[MT] (Constraint 12)	**7**	**7**	**9**	**9**	**7**
gdActionPointOffset^{Max}[MT] (Constraint 13)	**28**	**28**	**41**	**41**	**34**

Table B-20: Calculations for *gdActionPointOffset*.

In order to determine the configuration range of *gdActionPointOffset* an additional margin of safety is taken into account. As a result, the parameter *gdActionPointOffset* must be configurable over a range of 1 to 63 MT. This also allows a static slot design with additional safety margin, i.e. the action point of the static slots includes a safety margin beyond the achievable precision of the cluster.

B.4.9 gdMinislotActionPointOffset

Consider the following assumptions:

- The minislot action point *gdMinislotActionPointOffset* is greater than or equal to the assumed precision *aAssumedPrecision* reduced by the minimum propagation delay *adPropagationDelayMin* of the network.

Constraint 14:

gdMinislotActionPointOffset[MT] >= **ceil(** (*aAssumedPrecision*[µs] - *adPropagationDelayMin*[µs]) / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **)**

gdMinislotActionPointOffset shall be configurable in a range of 1 to 31 MT.

gdMinislotActionPointOffset can be independently configured from *gdActionPointOffset*. This is useful if the static segment design includes an additional safety margin that is not required in the dynamic segment. The independent choice of *gdMinislotActionPointOffset* allows increased throughput in the dynamic segment.

B.4.10 gdSymbolWindowActionPointOffset

Consider the following assumptions:

- The symbol window action point *gdSymbolWindowActionPointOffset* is greater than or equal to the assumed precision *aAssumedPrecision* reduced by the minimum propagation delay *adPropagationDelayMin* of the network.
- The symbol window action point *gdSymbolWindowActionPointOffset* is greater than or equal to the action point of the preceding segment.
- At least *gdWakeupRxIdle* bit times without activity on the bus shall take place before the start of a WUDOP transmission.

Constraint 15:

adSymbolWindowGuardInterval[MT] >= **ceil(** (*aAssumedPrecision*[µs] - *adPropagationDelayMin*[µs]) / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **)**

For the purposes of the calculation of parameter ranges it is assumed that *adSymbolWindowGuardInterval* takes on values within a range of 1 to 31 MT.

The following constraint shall be observed when MTS symbols may be transmitted within the symbol window.

Constraint 16:

gdSymbolWindowActionPointOffset[MT] = **max(** *adSymbolWindowGuardInterval*[MT]**;**

if(*gNumberOfMinislots* = 0**;** *gdActionPointOffset*[MT]**;** *gdMinislotActionPointOffset*[MT] **))**

The following constraint shall be observed when WUDOP symbols may be transmitted within the symbol window.

Constraint 17:

gdSymbolWindowActionPointOffset[MT] = **max(ceil(** (*gdWakeupRxIdle*[gdBit] *

adBitMax[µs/gdBit] - *adPropagationDelayMin*[µs] + *adMicrotickMaxDistError*[µs]) /

(*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **);** *adSymbolWindowGuardInterval***;**

if(*gNumberOfMinislots* = 0**;** *gdActionPointOffset*[MT]**;** *gdMinislotActionPointOffset*[MT] **))**

Systems that transmit both MTSs and WUDOPs shall observe both Constraint 16 and Constraint 17.

gdSymbolWindowActionPointOffset shall be configurable within a range of 1 to 63 MT.

gdSymbolWindowActionPointOffset can be configured to a value different than *gdActionPointOffset* and *gdMinislotActionPointOffset*. This is useful if the static segment design includes an additional safety margin that is not required in the symbol window. If a dynamic segment is present, *adSymbolWindowGuardInterval* should be chosen equal to *gdMinislotActionPointOffset*.

B.4.11 gdMinislot

The minislot consists of two parts:

- the part before the minislot action point
- the part after the minislot action point

Constraint 14 defines the part before the minislot action point.

For the part after the minislot action point the following two assumptions need to be considered:

1. The start of a dynamic frame transmission shall be recognized by all nodes within the same minislot.
2. The start of the idle phase as signalled by the *potential idle start* signal after the transition from the DTS low phase to the DTS high bit shall be recognized by all nodes within the same minislot.

To consider the first assumption, the part after the minislot action point shall be greater than or equal to the sum of:

- the assumed precision *aAssumedPrecision*
- the maximum propagation delay *adPropagationDelayMax* of the network
- the physical layer effects that lead to a length change of the transmitted TSS (as characterized by the parameters *dFrameTSSLengthChange* and *dFrameTSSEMIInfluence*).

See Figure 3-10 for additional details.

Constraint 18:

gdMinislot[MT] >= *gdMinislotActionPointOffset*[MT] + **ceil(** (*adPropagationDelayMax*[µs] +

aAssumedPrecision[µs] - **min$_{M,N}$(** $dFrameTSSLengthChange_{M,N}$[µs] +

$dFrameTSSEMIInfluence_{M,N}$[µs] **)**) / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **)**

To consider the second assumption, the part after the minislot action point shall be greater than or equal to the sum of:

- the assumed precision *aAssumedPrecision*
- the maximum propagation delay *adPropagationDelayMax* of the network
- the overall effect of ringing if present

Constraint 19:

gdMinislot[MT] >= *gdMinislotActionPointOffset*[MT] + **ceil(** (*adPropagationDelayMax*[µs] + *aAssumedPrecision*[µs] + *adMaxIdleDetectionDelayAfterHIGH*[µs]) / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **)**

With the assumptions for *gdMinislotActionPointOffset* given in section B.4.9, and allowing for a margin of safety, the parameter *gdMinislot* must be configurable over a range of 2 to 63 MT.

In addition to the previous constraints, it is possible that for certain system configurations the parameter *gdMinislot* has an additional constraint related to the parameter *gdIgnoreAfterTx*. Refer to section B.4.37 for additional details.

B.4.12 gdStaticSlot

Consider the following assumptions:

- A frame consists of at least *gdTSSTransmitter*, *cdFSS*, 80 gdBit of header and trailer (with *gPayloadLengthStatic* = 0), and *cdFES*.
- Each two-byte payload data word adds a duration equal to 2 * (8 gdBit + *cdBSS*[gdBit]) = 20 gdBit.

The length of a frame is given by:

[21] *aFrameLength*[gdBit] = *gdTSSTransmitter*[gdBit] + *cdFSS*[gdBit] + 80 gdBit + *aPayloadLength*[two-byte word] * 20 gdBit / two-byte word + *cdFES*[gdBit]

Substituting the length of a static frame for *aPayloadLength* results in:

[22] *aFrameLengthStatic* = *aFrameLength* with *aPayloadLength* = *gPayloadLengthStatic*

In addition to the frame length, the following effects must also be considered:

- The length of a static slot, expressed as a number of macroticks, depends on the maximum clock deviation (*gClockDeviationMax*) and the minimum duration of a macrotick.
- The effects of system precision may be taken into account by including *gdActionPointOffset* before and after frame transmission.
- An idle detection time of *cChannelIdleDelimiter* must be considered. For the purposes of this constraint, the idle detection time must be reduced by the duration of the high bit of the FES as this appears to the receiver as part of the idle detection.
- The maximum and minimum propagation delays of the cluster must also be taken into account.
- The effects of ringing must be taken into account by adding a parameter that will be zero if no ringing occurs.

Using equation [21] and [22] the minimum static slot length for systems can be calculated by Constraint 20:

Constraint 20:

gdStaticSlot[MT] >= 2 * *gdActionPointOffset*[MT] +

ceil(((*aFrameLengthStatic*[gdBit] - 0.5 * *cdFES*[gdBit] +

cChannelIdleDelimiter[gdBit]) * *adBitMax*[µs/gdBit] + *adPropagationDelayMin*[µs] +

adPropagationDelayMax[µs] + *adMaxIdleDetectionDelayAfterHIGH*[µs]) /

(*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **)**

Constraint 20 is valid for all systems. If, however, *gdActionPointOffset* is computed using Constraint 12 as an equality constraint (as opposed to using Constraint 12 as a "greater than or equal to" constraint, or some other constraint such as Constraint 13), the following constraint, which might result in a smaller required static slot size, should be used instead of Constraint 20:

Constraint 21:

gdStaticSlot[MT] >= *gdActionPointOffset*[MT] +

ceil(((*aFrameLengthStatic*[gdBit] - 0.5 * *cdFES*[gdBit] +

cChannelIdleDelimiter[gdBit]) * *adBitMax*[µs/gdBit] + *adPropagationDelayMax*[µs] +

aAssumedPrecision[µs] + *adMaxIdleDetectionDelayAfterHIGH*[µs]) /

(*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **)**

Bit Rate [Mbit/s]	**2.5**	**5**	**10**
gdActionPointOffset^{Min}[MT]	1		
aFrameLengthStatic^{Min}[gdBit]	84	84	84
adBitMax[µs]	0.4006	0.2003	0.10015
adPropagationDelayMax^{Min}[µs]	0.401	0.200	0.100
aAssumedPrecison^{Min}[µs][a]	0.701	0.351	0.351[b]
adMaxIdleDetectionDelayAfterHIGH^{Min}[µs]	0		
gdMacrotick^{Max}[µs]	6		
gdStaticSlot^{Min}[MT][c]	**8**	**5**	**3**
gdActionPointOffset^{Max}[MT]	63		
aFrameLengthStatic^{Max}[gdBit]	2628[d]	2631	2638
adBitMax[µs]	0.4006	0.2003	0.10015
adPropagationDelayMax^{Max}[µs]	3.051	2.775	2.638
adPropagationDelayMin^{Max}[µs]	2.999	2.750	2.625
adMaxIdleDetectionDelayAfterHIGH^{Max}[µs]	2.0756	1.8753	1.77515
gdMacrotick^{Min}[µs]	2	1	
gdStaticSlot^{Max}[MT][e]	**660**	**664**	**399**

Table B-21: Calculations for *gdStaticSlot*.

a. *aAssumedPrecision*^{Min} is equal to *aBestCasePrecision*, which is a function of bit rate as it limits the allowable choices for *adMicrotickMax*. See section B.4.2.4 for details.
b. This is the value of *aBestCasePrecision* consistent with *adMicrotickMax* that allows the use of maximum duration microticks, as this minimizes the value of *gdStaticSlot*.
c. Calculated using Constraint 21.
d. The calculations in the table are based on the transmission of the longest possible static frame, i.e., a static frame with a payload of *cPayloadLengthMax* two-byte words. Note that there are physical layer constraints that may prevent a frame of this length from being sent. See section B.4.40 for details.
e. Calculated using Constraint 20.

As a result, the parameter *gdStaticSlot* must be configurable over a range of 3 to 664 MT.

In addition to the previous constraints, it is possible that for certain system configurations the parameter *gdStaticSlot* has an additional constraint related to the parameter *gdIgnoreAfterTx*. Refer to section B.4.37 for additional details.

B.4.13 gdSymbolWindow

Consider the following assumptions:

- The length of an MTS symbol is defined by *cdCAS*[192].
- An MTS symbol is sent with a leading transmission start sequence. The symbol window takes this into account by using *gdTSSTransmitter* + *cdCAS*.
- The influence of the precision is taken into account by *gdSymbolWindowActionPointOffset*.
- After completion of the transmission of the symbol an idle detection time of *cChannelIdleDelimiter* is required. Idle detection can be delayed by ringing.
- The end of the symbol reception, as seen by the receiver, depends not only on the propagation delay inside the network but will be further delayed due to idle reaction times inside bus drivers (*dBDTxai*, *dBDRxai*) and active stars (*dStarSymbolEndLengthChange*) and the effects of EMI as given by $dSymbolEMIInfluence_{M,N}$.
- At least *gdActionPointOffset* macroticks must pass between the end of the symbol transmission (including the following idle detection time of *cChannelIdleDelimiter* bits) and the beginning of the static segment.
- The maximum time required to transmit the symbol is increased depending on the clock deviation (*adBitMax*).
- The duration of a macrotick may also be decreased depending on the clock deviation (*gClockDeviationMax*).
- The reception of the actively transmitted (i.e., HIGH and LOW) phases of the WUDOP must be completed, at the latest, by the end of the symbol window.
- The detection of the final high phase required by the wakeup decoding process (i.e., the phase that might be caused by the inactivity on the bus following the end of the WUDOP transmission) must be completed, at the latest, by the end of the NIT.

The following constraint shall be observed when MTS symbols may be transmitted within the symbol window.

[192] The collision avoidance symbol, CAS, is the same as the media access test symbol, MTS.

Constraint 22:

gdSymbolWindow[MT] = *gdSymbolWindowActionPointOffset*[MT] +
adSymbolWindowGuardInterval[MT] +
ceil((*adPropagationDelayMin*[µs] + *adPropagationDelayMax*[µs] +
(*gdTSSTransmitter*[gdBit] + *cdCAS*[gdBit] + *cChannelIdleDelimiter*[gdBit]) *
adBitMax[µs/gdBit] + (**max$_{M,N}$(** *anRingPath$_{M,N}$* **)** * *dRing*[µs]) +
max$_{M,N}$(*dSymbolEMIInfluence$_{M,N}$*[µs] + *dBDTxai*[µs] + *dBDRxai*[µs] +
nStarPath$_{M,N}$ * *dStarSymbolEndLengthChange*[µs] **)** +
adMicrotickMaxDistError[µs]) / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) +
max(0; *gdActionPointOffset*[MT] - *gdNIT*[MT] - *adSymbolWindowGuardInterval*[MT] +
(*aOffsetCorrectionMax*[µs] - *adPropagationDelayMin*[µs] +
adMicrotickMaxDistError[µs]) / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **))**

The following constraint shall be observed when WUDOP symbols may be transmitted within the symbol window.

Constraint 23:

gdSymbolWindow[MT] = *gdSymbolWindowActionPointOffset*[MT] +
adSymbolWindowGuardInterval[MT] +
ceil((*adPropagationDelayMin*[µs] + *adPropagationDelayMax*[µs] +
(5 * *gdWakeupTxActive*[gdBit] + *cChannelIdleDelimiter*[gdBit]) * *adBitMax*[µs/gdBit] +
adMaxIdleDetectionDelayAfterHIGH[µs] + *adMicrotickMaxDistError*[µs]) /
(*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) +
max(0**;** *gdActionPointOffset*[MT] - *adSymbolWindowGuardInterval*[MT] - *gdNIT*[MT] +
(*aOffsetCorrectionMax*[µs] - *adPropagationDelayMin*[µs] +
adMicrotickMaxDistError[µs]) / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*))**;**
((*gdWakeupRxIdle*[gdBit] - *cChannelIdleDelimiter*[gdBit]) * *adBitMax*[µs/gdBit] +
aOffsetCorrectionMax[µs] + *adMicrotickMaxDistError*[µs]) /
(*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) - *gdNIT*[MT] **))**

Systems that transmit both MTSs and WUDOPs shall observe both Constraint 22 and Constraint 23.

For the purposes of determining the configurable range of this parameter, it can be realized that the transmission time of a WUDOP is much larger than the transmission time of an MTS. As a result, the maximum range of the parameter will be determined by Constraint 23 only.

Bit Rate [Mbit/s]	2.5	5	10
gdSymbolWindowActionPointOffset^{Max}[MT]	63		
adSymbolWindowGuardInterval^{Max}[MT]	31		
gdActionPointOffset^{Max}[MT]	63		

Table B-22: Calculations for *gdSymbolWindow*.

Bit Rate [Mbit/s]	2.5	5	10
gdWakeupTxActive[gdBit]	15	30	60
gdWakeupRxIdle[gdBit]	59		
gdNIT^{Min}[MT]	2		
aOffsetCorrectionMax[a][µs]	0.9955	0.49775	0.49775
adBitMax[µs]	0.4006	0.2003	0.10015
adMaxIdleDetectionDelayAfterHIGH^{Max}[µs]	2.0756	1.8753	1.77515
adPropagationDelayMin^{Max}[µs]	2.999	2.750	2.625
adPropagationDelayMax^{Max}[µs]	3.051	2.775	2.638
gdMacrotick^{Min}[µs]	2	1	
gdSymbolWindow^{Min}[MT]	**0**[b]		
gdSymbolWindow^{Max}[MT]	**145**	**162**	**161**

Table B-22: Calculations for *gdSymbolWindow*.

a. It is the maximum possible value if *gdNIT* is a minimum.
b. If no symbol window is used at all *gdSymbolWindow* would be configured to 0 MT.

As a result, the parameter *gdSymbolWindow* must be configurable over a range of 0 to 162 MT.

In addition to the previous constraints, it is possible that for certain system configurations the parameter *gdSymbolWindow* has an additional constraint related to the parameter *gdIgnoreAfterTx*. Refer to section B.4.37 for additional details.

Note - [EPL10] allows the possibility of using an active star device to play the role of a BD in a FlexRay system (i.e., from the physical layer perspective the transmission can originate and / or terminate in the active star). For reasons of clarity, the constraints in this section do not comprehend this (i.e., the constraints assume the transmission originates or terminates in a bus driver). For systems that utilize the active stars as transceivers it is necessary to modify the constraints to make appropriate use of active star-related parameters instead of BD-related parameters (for example, to use *dStarRxai* instead of *dBDRxai*).

B.4.14 gMacroPerCycle

The number of macroticks per cycle is based on the cycle duration and the macrotick length.

Constraint 24:

gdMacrotick[µs] = *gdCycle*[µs] / *gMacroPerCycle*

with *gdCycle*[µs] <= *cdCycleMax*[µs] and *gdMacrotick*[µs] >= *cdMinMTNom*[µs]. Note that *gMacroPerCycle*[MT] must be an integer value.

Bit Rate [Mbit/s]	2.5	5	10
gdCycle^{Max}[µs] = *cdCycleMax*	16000		
gdMacrotick^{Min}[µs]	2	1	
gMacroPerCycle^{Max}[MT]	**8000**	**16000**	**16000**

Table B-23: Calculations for *gMacroPerCycle*.

The cycle length in macroticks must also be equivalent to the sum of the lengths of the segments that make up the cycle.

Constraint 25:

gMacroPerCycle[MT] = *gdStaticSlot*[MT] * *gNumberOfStaticSlots* + *adActionPointDifference*[MT] + *gdMinislot*[MT] * *gNumberOfMinislots* + *gdSymbolWindow*[MT] + *gdNIT*[MT]

adActionPointDifference[MT] is introduced in B.4.17 and calculated in equation [25].

Bit Rate [Mbit/s]	2.5	5	10
$gdStaticSlot^{Min}$[MT]	8	5	3
$gNumberOfStaticSlots^{Min}$	2		
$adActionPointDifference^{Min}$[MT]	0		
$gdMinislot^{Min}$[MT]	2		
$gNumberOfMinislots^{Min}$[Minislot]	0		
$gdSymbolWindow^{Min}$[MT]	0		
$gdNIT^{Min}$[MT]	2		
$gMacroPerCycle^{Min}$[MT]	**18**	**12**	**8**

Table B-24: Calculations for *gMacroPerCycle*.

As a result, the parameter *gMacroPerCycle* must be configurable over a range of 8 to 16000 MT.

B.4.15 pMicroPerCycle

The cycle length in microticks is calculated using the following equations:

Constraint 26:

pMicroPerCycle[µT] = **round(** *gdCycle*[µs] / *pdMicrotick*[µs/µT] **)**

pMicroPerCycle is always a positive integer number.

In order to define a minimum parameter range that an implementation must support, the minimum number of microticks in a cycle is determined under the following assumptions:

- the minimum cycle consists of two static slots plus the NIT,
- the minimum static slot can be calculated using Constraint 21,
- the maximum microtick length is used.

With these assumptions a lower bound can be calculated by using following equations:

[23] $pMicroPerCycle^{Min}$[µT]= **round(** (2 * *gdStaticSlot*[MT] + *gdNIT*[MT]) * (*gdMacrotick*[µs/MT] / *pdMicrotick*[µs/µT]) **)**

with *gdStaticSlot*[MT] equal to Constraint 21:

gdStaticSlot[MT] = *gdActionPointOffset*[MT] +

ceil(((*aFrameLengthStatic*[gdBit] - 0.5 * *cdFES*[gdBit] +

cChannelIdleDelimiter[gdBit]) * *adBitMax*[µs/gdBit] + *adPropagationDelayMax*[µs] +

aAssumedPrecision[µs] + *adMaxIdleDetectionDelayAfterHIGH*[µs]) /

(*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **)**

Bit Rate [Mbit/s]	2.5	5	10
gdActionPointOffset^{Min}[MT]	1		
gdNIT^{Min}[MT]	2		
gdMacrotick[µs]	2.05[a]	2	1
aFrameLengthStatic^{Min}[gdBit]	84		
aAssumedPrecision^{Min}[µs] = *aBestCase-Precision*^{Min} @ *pdMicrotick*^{Max}[µs]	0.701		0.351
adBitMax[µs]	0.4006	0.2003	0.10015
gdStaticSlot^{Min}[MT]	20	11	11
adPropagationDelayMax^{Min}[µs]	0.401	0.200	0.100
adMaxIdleDetectionDelayAfterHIGH^{Min}[µs]	0		
pdMicrotick^{Max}[µs]	0.05		0.025
pMicroPerCycle^{Min}[µT]	**1722**	**960**	**960**

Table B-25: Calculations for *pMicroPerCycle*.

a. *gdMacrotick* is chosen in a way that *pMicroPerCycle* results in a minimum cycle length.

Bit Rate [Mbit/s]	2.5	5		10	
***pdMicrotick*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
gdCycle^{Max}[µs] = *cdCycleMax*	16000				
pMicroPerCycle^{Max}[µT]	**320000**	**320000**	**640000**	**640000**	**1280000**

Table B-26: Calculations for *pMicroPerCycle*.

As a result, the parameter *pMicroPerCycle* must be configurable over a range of 960 to 1280000 µT.

B.4.16 gdDynamicSlotIdlePhase

Consider the following assumptions:

- The duration of *gdDynamicSlotIdlePhase*[Minislot] must be greater than or equal to the idle detection time, which can be delayed by ringing.
- The idle detection time must be calculated based on the uncorrected bit time and therefore equals *cChannelIdleDelimiter* * *adBitMax*.
- The macroticks may also be shortened by the clock deviation.

Constraint 27:

gdDynamicSlotIdlePhase[Minislot] >= **ceil(** (**ceil(** (*cChannelIdleDelimiter* * *adBitMax*[µs] + *adMaxIdleDetectionDelayAfterHIGH*[µs] + *aAssumedPrecision*[µs] + *adPropagationDelayMax*[µs]) / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **)** - (*gdMinislot*[MT] - *gdMinislotActionPointOffset*[MT])) / *gdMinislot*[MT/Minislot] **)**

The minimum value for *gdDynamicSlotIdlePhase* would occur for large values of *gdMinislot* combined with small values of *gdMinislotActionPointOffset*. In these cases Constraint 27 will evaluate to zero.

The maximum required value for *gdDynamicSlotIdlePhase* cannot easily be determined from Constraint 27 by simply using the maximum or minimum values of the input parameters. The worst case situation would arise when some nodes in the system detect the last potential idle start event just before the end of its local minislot. In this case, *gdDynamicSlotIdlePhase* would need to be large enough to ensure that the CHIRP is detected during the dynamic slot idle phase, and thus

[24] *gdDynamicSlotIdlePhase*^{Max}[Minislot] = **ceil(** (*cChannelIdleDelimiter* * *adBitMax*[µs] + *adMicrotickMaxDistError*[µs]) / (*gdMacrotick*[µs/MT] * (*gdMinislot*[MT/Minislot] / (1 + *gClockDeviationMax*))) **)**

Bit Rate [Mbit/s]	**2.5**	**5**		**10**	
***adMicrotickMax*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
adBitMax[µs]	0.4006	0.2003		0.10015	
gdMacrotick^{Min}[µs]	2	2	1	1	1
gdMinislot^{Min}[MT]	2	2	2	2	2
gdDynamicSlotIdlePhase^{Min}[Minislot]	**0**	**0**	**0**	**0**	**0**
gdDynamicSlotIdlePhase^{Max}[Minislot]	**2**	**1**	**2**	**1**	**1**

Table B-27: Calculations for *gdDynamicSlotIdlePhase*.

As a result, the parameter *gdDynamicSlotIdlePhase* must be configurable over a range of 0 to 2 minislots.

B.4.17 gNumberOfMinislots

Consider the following:

[25] *adActionPointDifference*[MT] = **if(or(** *gdActionPointOffset* <= *gdMinislotActionPointOffset***;** *gNumberOfMinislots* = 0**);** 0**;** *gdActionPointOffset* - *gdMinislotActionPointOffset* **)**

Constraint 28:

gNumberOfMinislots[Minislot] = (*gMacroPerCycle*[MT] - *gdNIT*[MT] - *adActionPointDifference*[MT] - *gNumberOfStaticSlots* * *gdStaticSlot*[MT] - *gdSymbolWindow*[MT]) / *gdMinislot*[MT/Minislot]

gNumberOfMinislots is always an integer. To fulfill Constraint 28 the parameters on the right side of the equation must be chosen so that *gNumberOfMinislots* results in an integer.[193]

Bit Rate [Mbit/s]	2.5	5	10
gMacroPerCycle^{Max}[MT]	8000	16000	
gdNIT^{Min}[MT]	2		
adActionPointDifference^{Min}[MT]	0		
gNumberOfStaticSlots^{Min}	2		
gdStaticSlot^{Min}[MT] with *gdMacrotick* = 2 µs	21	-	-
gdStaticSlot^{Min}[MT] with *gdMacrotick* = 1 µs	-	21	11
gdSymbolWindow^{Min}[MT]	0		
gdMinislot^{Min}[MT]	2		
gNumberOfMinislots^{Max}[Minislot]	**3978**	**7978**	**7988**

Table B-28: Calculations for *gNumberOfMinislots*.

If no dynamic segment is used, *gNumberOfMinislots* is set to zero.

As a result, the parameter *gNumberOfMinislots* must be configurable over a range of 0 to 7988 minislots.

To estimate the number of frames which could be transmitted in the dynamic segment the following equation may be useful:

[26] *aMinislotPerDynamicFrame*[Minislot] = 1 Minislot +

ceil(((*aFrameLengthDynamic*[gdBit] + *adDTSLow*^{Min}[gdBit]) * *adBitMax*[µs/gdBit] + *adMicrotickMaxDistError*[µs]) / ((*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) * *gdMinislot*[MT/Minislot]) **)** + *gdDynamicSlotIdlePhase*[Minislot]

with

[27] *adDTSLow*^{Min} = 1 gdBit

B.4.18 pRateCorrectionOut

Consider the following assumptions:

- The rate correction mechanism must compensate the accumulated error in microticks of one complete cycle.
- The error of one cycle arises from worst-case clock deviations and is limited to twice the maximum deviation of the clock frequency *gClockDeviationMax*.

Depending on, for example, the implementation of the external rate/offset correction, the value of the *pExternRateCorrection* parameter might influence the choice of this parameter value as well.[194] Detailed

[193] This can be accomplished by e.g. increasing *gdNIT* or decreasing *gMacroPerCycle*.

[194] The system designer must ensure that the cluster will never be operated at an aggregate rate (i.e., the internal rate corrected by the external rate) that exceeds the value of the clock deviation used in the system design.

analysis of effects due to external clock correction terms might influence the parameter range as well. In all cases, however, the following constraint must be fulfilled:

Constraint 29:

pRateCorrectionOut[µT] = **ceil(** *pMicroPerCycle*[µT] * 2 * *gClockDeviationMax* *

(1 + *gClockDeviationMax*) **)**

Note that Constraint 29 does not include the effects of some influences on the determination of the rate correction terms that, for some systems, might result in a clock correction value a few microticks larger or smaller than the ideal value. Examples of such effects include quantization and inaccuracies in the measurement of the time reference point due to EMI or glitches. These effects are to some extent compensated by the cluster drift damping mechanism, which tends to reduce their impact.

Bit Rate [Mbit/s]	2.5	5	10
$pMicroPerCycle^{Min}$[µT]	1722	960	960
$pRateCorrectionOut^{Min}$[µT]	**6**	**3**	**3**

Table B-29: Calculations for *pRateCorrectionOut*.

***pdMicrotick*[µs]**	**0.050**	**0.025**	**0.0125**
$pMicroPerCycle^{Max}$[µT]	320000	640000	1280000
$pRateCorrectionOut^{Max}$[µT]	**962**	**1923**	**3846**

Table B-30: Calculations for *pRateCorrectionOut*.

As a result *pRateCorrectionOut* must be configurable over a range of 3 to 3846 µT.

B.4.19 Offset Correction

B.4.19.1 aOffsetCorrectionMax

The parameter *aOffsetCorrectionMax* represents the maximum amount of offset correction that would be required in a properly working system. *aOffsetCorrectionMax* also includes the amount of offset correction based on an external offset correction.

Consider the following assumptions:

- The clocks of two nodes may have an offset of up to the assumed precision.
- The external offset correction might need to be applied.
- The received frames might be received with the maximum propagation delay of which only the minimal possible propagation delay is compensated by the local delay compensation.

These assumptions are essentially identical to the assumptions on *adInitializationErrorMax* (see section B.4.5), and as a result the following constraint must be fulfilled:

Constraint 30:

aOffsetCorrectionMax[µs] >= *adInitializationErrorMax*[µs]

Constraint 30 provides only a lower bound on the value of *aOffsetCorrectionMax*. In order to compute the configuration ranges of parameters dependent on *aOffsetCorrectionMax* it is also necessary to have an assumed upper bound. This can be computed by making a structural argument based on the maximum

measured offset values possible given the parameters that determine the characteristics of the static slot (i.e., *gdStaticSlot* and *gdActionPointOffset*).

As the offset correction term can be either negative or positive, it is necessary to distinguish between the maximum negative and maximum positive value from which the maximum absolute value is taken. Therefore two auxiliary variables are introduced: *aNegativeOffsetCorrectionMax* and *aPositiveOffsetCorrectionMax*. To determine the maximum offset correction value only the absolute value of the negative offset correction is considered, which is reflected in the equation below. Therefore the equation for *aNegativeOffsetCorrectionMax* is formulated such that is results in a positive value.

Consider the following assumptions for deriving the maximum possible negative offset correction:

- The secondary time reference point of the incoming frame is synchronous with the slot boundary.[195]
- The maximum possible delay compensation and decoding correction is applied.
- The maximum possible external offset correction is considered.

[28] *aNegativeOffsetCorrectionMax*[µs] = *gdActionPointOffset*[MT] * (*gdMacrotick*[µs/MT] / (1 - *gClockDeviationMax*)) + **max$_N$(** *adMicrotickDistError$_N$*[µs] + (**max(** *pDelayCompensation[A]$_N$*[µT] **;** *pDelayCompensation[B]$_N$*[µT] **)** + *pDecodingCorrection$_N$*[µT]) * (*pdMicrotick$_N$*[µs/µT] / (1 - *gClockDeviationMax*)) **)** + *gExternOffsetCorrection*[µs]

Consider the following assumptions for deriving the maximum possible positive offset correction:

- The frame decoded signal of the incoming sync frame arrives synchronous with the slot boundary.[196]
- There are exactly 80 plus 20 times *gPayloadLengthStatic* bits between the secondary time reference point and the frame decoded signal.
- The received frame is transmitted with the minimum allowed bit length.
- The received frame is subjected to minimum delay compensation and decoding correction.

[29] *aPositiveOffsetCorrectionMax*[µs] = (*gdStaticSlot*[MT] - *gdActionPointOffset*[MT]) * (*gdMacrotick*[µs/MT] / (1 - *gClockDeviationMax*)) - (8 + 2 * *gPayloadLengthStatic*) * 10[gdBit] * *adBitMin*[µs/gdBit] - **min$_N$(** -*adMicrotickDistError$_N$*[µs] + (**min(** *pDelayCompensation[A]$_N$*[µT] **;** *pDelayCompensation[B]$_N$*[µT] **)** + *pDecodingCorrection$_N$*[µT]) * (*pdMicrotick$_N$*[µs/µT] / (1 + *gClockDeviationMax*)) **)** + *gExternOffsetCorrection*[µs]

Using these auxiliary variables, it is possible to derive a maximal absolute value for offset correction:

[30] *aOffsetCorrectionMax*[µs] <= **max(** *aNegativeOffsetCorrectionMax*[µs] **;** *aPositiveOffsetCorrectionMax*[µs] **)**

[195] This is actually impossible as such a frame would be discarded by the FSP.
[196] This causes a boundary violation, but the frame is still used for clock synchronization.

Bit Rate [Mbit/s]	2.5	5		10	
***adMicrotickMax*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
adInitializationErrorMax^{Min}[µs]	0.701	0.701	0.351	0.351	0.175
aOffsetCorrectionMax^{Min}[µs]	**0.701**	**0.701**	**0.351**	**0.351**	**0.175**
gdMacrotick^{Max}[µs]	6	6	6	6	3
gdActionPointOffset^{Max}[MT]	63				
gdStaticSlot^{Max}[MT][a]	304	216		172	217
gPayloadLengthStatic^{Max}[two-byte word][b]	127[c]				
adBitMin[µs]	0.3994	0.1997		0.09985	
pDelayCompensation[Ch]^{Min}[µT]	7	4	7	4	7
pDelayCompensation[Ch]^{Max}[µT]	61	55	111	105	211
pDecodingCorrection^{Min}[µT]	24	12	24	12	24
pDecodingCorrection^{Max}[µT]	56	40	80	68	136
pdMicrotick[µs]	0.050	0.050	0.025	0.025	0.0125
gExternOffsetCorrection^{Max}[µs]	0.35				
aNegativeOffsetCorrectionMax^{Max}[µs]	**384.827**	**383.725**	**383.725**	**383.274**	**193.990**
aPositiveOffsetCorrectionMax^{Max}[µs]	**400.597**	**395.766**	**395.766**	**393.351**	**201.063**
aOffsetCorrectionMax^{Max}[µs]	**400.597**	**395.766**	**395.766**	**393.351**	**201.063**

Table B-31: Calculations for *aOffsetCorrectionMax*.

a. This value was derived just as in Table B-21, but with a value of 6, respectively 3, for *gdMacrotick*.
b. The largest frame size is assumed as this results in the largest possible discrepancy between nominal and actual frame length.
c. For 2.5 Mbit/s the calculations in this table are conservative, as payload length is restricted by the maximum transmission duration *adTxMax*. See section B.4.40 for details.

B.4.19.2 pOffsetCorrectionOut

Since the check for the offset correction is applied before the external offset correction is included, the effects of the external offset correction that are included in the definition of *aOffsetCorrectionMax* must be removed when calculating *pOffsetCorrectionOut*.

Constraint 31:

pOffsetCorrectionOut[µT] = **ceil(** (*aOffsetCorrectionMax*[µs] - *gExternOffsetCorrection*[µs]) / (*pdMicrotick*[µs/µT] / (1 + *gClockDeviationMax*)) **)**

Bit Rate [Mbit/s]	2.5	5		10	
***pdMicrotick*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
aOffsetCorrectionMax^{Min}[µs]	0.701	0.701	0.351	0.351	0.175
aOffsetCorrectionMax^{Max}[µs]	400.597	395.766	395.766	393.351	193.990
gExternOffsetCorrection^{Min}[µs]	0				
gExternOffsetCorrection^{Max}[µs]	0.35				
pOffsetCorrectionOut^{Min}[µT]	**15**	**15**	**15**	**15**	**15**
pOffsetCorrectionOut^{Max}[µT]	**8017**	**7921**	**15841**	**15744**	**16082**

Table B-32: Calculations for *pOffsetCorrectionOut*.

As a result, the parameter *pOffsetCorrectionOut* shall be configurable over a range of 15 to 16082 µT.

B.4.20 pOffsetCorrectionStart

Consider the following assumptions:

- The offset correction phase starts at *pOffsetCorrectionStart*.
- The offset correction phase ends at the end of the cycle.

Thus it holds that

Constraint 32:

pOffsetCorrectionStart[MT] = *gMacroPerCycle*[MT] - *adOffsetCorrection*[MT]

with *adOffsetCorrection* the length of the offset correction phase, which is constrained in equation [33] in the next section. This parameter should be set to the same value for all nodes in a cluster.[197]

B.4.21 gdNIT

Consider the following assumptions:

1. The NIT consists of offset calculation phase and offset correction phase.
2. The duration of offset calculation may vary from implementation to implementation, and may vary depending on the number of received sync frames. The offset calculation must be completed before the start of the offset correction phase. The upper limit for the duration of the offset calculation is defined in section 8.6.2. The offset correction calculation must be completed no later than *cdMaxOffsetCalculation* after the end of the static segment or 1 MT after the start of the NIT, whichever occurs later.
3. The earliest start of the offset correction phase is 1 MT after the start of the NIT.
4. The duration of offset correction phase is at least 1 MT.
5. The maximum possible value for offset correction is *aOffsetCorrectionMax*. The offset correction phase must be long enough (i.e., contain enough macroticks) to correct this amount of offset while still keeping the length of shortened macroticks greater than or equal to *cMicroPerMacroMin* microticks.

[197] For TT-E systems it may be beneficial to allow different offset correction starts for the coldstart and non-sync nodes (see section B.7.9), and as a result this is a p-parameter rather than a g-parameter.

6. The offset correction phase and the offset calculation phase can be overlaid with parts of the rate calculation phase.
7. The duration of rate calculation may vary from implementation to implementation, and may vary depending on the number of received sync frames. The rate calculation must be completed before the end of the NIT. The upper limit for the duration of the rate calculation is defined in section 8.6.3. The rate correction calculation must be completed no later than *cdMaxRateCalculation* after the end of the static segment or 2 MT after the start of the NIT, whichever occurs later. This can induce the need for a prolonged NIT.

Due to item 1, 6 and 7, the NIT length *gdNIT* is either the remaining time required to calculate the offset correction and then execute it, or the remaining time required to ensure that rate calculation finishes before the cycle ends, whichever takes longer. Thus *gdNIT* can be constrained by

Constraint 33:

gdNIT[MT] >= **max(** *adRemRateCalculation*[MT]**;** *adRemOffsetCalculation*[MT] + *adOffsetCorrection* [MT] **)**

Due to item 2 the remaining length of the offset calculation phase during the NIT *adRemOffsetCalculation* can be defined by

[31] *adRemOffsetCalculation*[MT] <= **max(** 1**; ceil(** (*cdMaxOffsetCalculation*[µT] * *adMicrotickMax*[µs/µT] / (1 - *gClockDeviationMax*) + *adMicrotickMaxDistError*[µs] - (*adActionPointDifference*[MT] + *gdMinislot*[MT] * *gNumberOfMinislots* + *gdSymbolWindow*[MT]) * (*gdMacrotick*[µs] / (1 + *gClockDeviationMax*))) / (*gdMacrotick*[µs] / (1 + *gClockDeviationMax*)) **))**

[32] *adRemOffsetCalculation*[MT] >= 1

With item 5 *adOffsetCorrection*, the length of the offset correction phase, can be constrained by

[33] *adOffsetCorrection*[MT] >= **ceil(** (*aOffsetCorrectionMax*[µs] + *adMicrotickMaxDistError*[µs]) / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*) - *adMicrotickMax*[µs/µT] * *cMicroPerMacroMin*[µT/MT] / (1 - *gClockDeviationMax*)) **)**

Finally assumption 7 helps to define *adRemRateCalculation*, the time required to complete the remaining rate calculation after the beginning of the NIT. The rate calculation has to be ready early enough that it will finish before the cycle start even if a maximum negative offset correction of *aOffsetCorrectionMax* is applied, effectively adding *aOffsetCorrectionMax* to the required calculation time. It is therefore constrained by

[34] *adRemRateCalculation*[MT] <= **max(** 1**; ceil(** (*cdMaxRateCalculation*[µT] * *adMicrotickMax*[µs/µT] / (1 - *gClockDeviationMax*) + *adMicrotickMaxDistError*[µs] - (*adActionPointDifference*[MT] + *gdMinislot*[MT] * *gNumberOfMinislots* + *gdSymbolWindow*[MT]) * *gdMacrotick*[µs] / (1 + *gClockDeviationMax*) + *aOffsetCorrectionMax*[µs]) / (*gdMacrotick*[µs] / (1 + *gClockDeviationMax*)) **))**

Normally there is a period of inactivity following the end of transmission in a static slot and the start of the next static slot. If the cycle contains a dynamic segment, however, the period following the end of transmission of the dynamic segment and the beginning of the first static slot consists only of the period after the dynamic transmission ends until the end of the dynamic segment, the duration of a symbol window (if any), and the duration of the NIT (which can be shortened by a negative offset correction). If it is desired that all static slots should have a certain minimum margin between the end of the previous transmission and the beginning of the static slot it may be necessary to use a larger network idle time than would normally be required. For example, if Constraint 20 (which assumes the "space" at the end of a static slot is approximately equal to *gdActionPointOffset*) is used, the following equation would ensure that there is sufficient space between the end of a dynamic transmission and the beginning of the first static slot:

[35] *gdNIT*[MT] >= *gdActionPointOffset*[MT] - *gdSymbolWindow*[MT] - (*gdMinislot*[MT] - *gdMinislotActionPointOffset*[MT] + *gdDynamicSlotIdlePhase*[Minislot] * *gdMinislot*[MT/Minislot]) + **ceil(** (*aOffsetCorrectionMax*[µs] + *adMicrotickMaxDistError*[µs]) / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **)**

This equation would only apply for cycles that actually contain a dynamic segment. Obviously, if a different amount of "space" is reserved at the end of a static slot (for example, if *gdStaticSlot* is set using Constraint 21 instead of Constraint 20, or even a larger margin than implied by Constraint 20) equation [35] could be modified to provide a larger or smaller margin.[198] If the cycle does not contain a dynamic segment there is no need for an additional equation limiting the minimum size of the NIT.

The configurable minimum for *gdNIT* can be found by making the assumption that the NIT consists of only 1 MT for offset calculation and only 1 MT for offset correction. The configurable maximum for *gdNIT* can be found by making the assumption that the cycle consist of the largest possible number of macroticks and the NIT is the duration of the cycle minus the duration of two of the shortest static slots possible using the smallest possible nominal macrotick duration.

As a result, the parameter *gdNIT* must be configurable over a range of 2 MT to 15978 MT.

B.4.22 pExternRateCorrection

Consider the following assumption:

- The absolute value of the external rate correction value shouldn't be greater than the maximum acceptable rate correction value *pRateCorrectionOut*.

Constraint 34:

pExternRateCorrection[µT] <= *pRateCorrectionOut*[µT]

The application of external rate or offset correction is controlled by the hosts, and should be synchronized such that all nodes apply the same value at the same time in the same direction. In order to achieve this synchronization between the hosts is necessary.

$gExternRateCorrection^{Max}$[µs] is the maximum external rate correction value applied in a cluster. *gExternRateCorrection* has to be chosen in a way that for every microtick duration used in the cluster the resulting *pExternRateCorrection* values are integer numbers.[199]

[198] If the symbol window is used for transmission of MTS's or WUDOP's Constraints 22 and 23 will ensure the same property and an equation similar to equation [35] is not necessary.

[199] The ability to actually apply an external rate correction is affected by the cluster drift damping. The cluster drift damping could even be set to zero for systems where an external rate correction serves to prevent large deviations of the operating rate of the cluster.

Constraint 35:

pExternRateCorrection[µT] = *gExternRateCorrection*[µs] / *pdMicrotick*[µs/µT]

Bit Rate [Mbit/s]	2.5	5		10	
***adMicrotickMax*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
gExternRateCorrection^{Min}[µs]	0				
gExternRateCorrection^{Max}[µs]	0.35				
pdMicrotick^{Min}[µs]	0.050	0.025	0.025	0.0125	0.0125
pExternRateCorrection^{Min}[µT]	**0**				
pExternRateCorrection^{Max}[µT]	**7**	**14**	**14**	**28**	**28**

Table B-33: Calculations for *pExternRateCorrection*.

The range of *pExternRateCorrection* shall be configurable over a range of 0 to 28 µT.[200]

B.4.23 pExternOffsetCorrection

Consider the following assumption:

- The absolute value of the external offset correction value shouldn't be greater than the maximum acceptable offset correction value *pOffsetCorrectionOut*.

gExternOffsetCorrection^{Max}[µs] is the maximum external offset correction value applied in a cluster. *gExternOffsetCorrection* has to be chosen in a way that for every microtick duration used in the cluster the resulting *pExternOffsetCorrection* values are integers.

Constraint 36:

pExternOffsetCorrection[µT] = *gExternOffsetCorrection*[µs] / *pdMicrotick*[µs/µT]

Bit Rate [Mbit/s]	2.5	5		10	
***adMicrotickMax*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
gExternOffsetCorrection^{Min}[µs]	0				
gExternOffsetCorrection^{Max}[µs]	0.35				
pdMicrotick^{Min}[µs]	0.050	0.025	0.025	0.0125	0.0125
pExternOffsetCorrection^{Min}[µT]	**0**				
pExternOffsetCorrection^{Max}[µT]	**7**	**14**	**14**	**28**	**28**

Table B-34: Calculations for *pExternOffsetCorrection*.

The range of *pExternOffsetCorrection* shall be configurable over a range of 0 to 28 µT.[201]

[200] A small value for *pExternRateCorrection* (much smaller than *pRateCorrectionOut*) has the advantage that a node which does not apply the external rate correction due to a fault stays synchronized with the other nodes in the cluster.

[201] A small value for *pExternOffsetCorrection* (much smaller than *pOffsetCorrectionOut*) has the advantage that a node which does not apply the external offset correction due to a fault stays synchronized with the other nodes in the cluster.

B.4.24 pdListenTimeout

To configure the parameter *pdListenTimeout* the following constraint must be taken into account:

Constraint 37:

pdListenTimeout[µT] = 2 * (*pMicroPerCycle*[µT] + *pRateCorrectionOut*[µT])

***pdMicrotick*[µs]**	**0.050**	**0.025**	**0.0125**
pMicroPerCycle^{Max}[µT]	320000	640000	1280000
pRateCorrectionOut^{Max}[µT]	962	1923	3846
pdListenTimeout^{Max}[µT]	**641924**	**1283846**	**2567692**

Table B-35: Calculations for *pdListenTimeout*.

Bit Rate [Mbit/s]	**2.5**	**5**	**10**
pMicroPerCycle^{Min}[µT]	1722	960	960
pRateCorrectionOut^{Min}[µT]	6	3	3
pdListenTimeout^{Min}[µT]	**3456**	**1926**	**1926**

Table B-36: Calculations for *pdListenTimeout*.

As a result, the parameter *pdListenTimeout* shall be configurable over a range of 1926 to 2567692 µT.

B.4.25 pDecodingCorrection

Consider following assumption:

- The time difference between the secondary time reference point and the primary time reference point is the summation of *pDecodingCorrection* and *pDelayCompensation* (see Figure 3-11).

Constraint 38:

pDecodingCorrection[µT] = **round(** ((*gdTSSTransmitter*[gdBit] + *cdFSS*[gdBit] + 0.5 * *cdBSS*[gdBit]) * *cSamplesPerBit*[samples/gdBit]) / *pSamplesPerMicrotick*[samples/µT] **)**

Bit Rate [Mbit/s]	**2.5**	**5**		**10**	
***pdMicrotick*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
gdTSSTransmitter^{Min}[gdBit]	1				
gdTSSTransmitter^{Max}[gdBit]	5	8		15	
pSamplesPerMicrotick	1	2	1	2	1
pDecodingCorrection^{Min}[µT]	**24**	**12**	**24**	**12**	**24**
pDecodingCorrection^{Max}[µT]	**56**	**40**	**80**	**68**	**136**

Table B-37: Calculations for *pDecodingCorrection*.

As a result, the parameter *pDecodingCorrection* shall be configurable between 12 and 136 µT.

B.4.26 pDelayCompensation

Consider the following assumption:

- For the minimum propagation delay the general internal delay needs to be replaced by the specific one of the device in question.

Therefore the following constraints must be taken into account:

Constraint 39:

pDelayCompensation[Ch][µT] >= **ceil(** (*adPropagationDelayMin[Ch]*[µs] +

(*adInternalRxDelay*[samples] - **min$_N$(** *adInternalRxDelay$_N$*[samples] **)**) *

gdSampleClockPeriod[µs/samples]) / *pdMicrotick*[µs/µT] **)**

Constraint 40:

pDelayCompensation[Ch][µT] <= **floor(** *adPropagationDelayMax[Ch]*[µs] / *pdMicrotick*[µs/µT] **)**

In the absence of information on the specific distributions of the propagation delays in the cluster, it is recommended to set this parameter to a value corresponding to the lower value of Constraint 39. In case that not all communication controllers use the same microtick length, care should be taken that all communication controllers use effectively the same value in real time (for example, by choosing a slightly larger value that is evenly divisible by all occurring microtick lengths in the cluster).

Using detailed knowledge about the topology of the cluster for the configuration of *pDelayCompensation*, an improvement in both precision and NIT length is possible; the details are however beyond the scope of this specification.[202]

The values and constraints given in this appendix for precision and NIT assume an identical value in real time for all nodes of the cluster, i.e., the product of a node's configuration of *pDelayCompensation* and the node's microtick duration is the same for all nodes in the cluster.

Bit Rate [Mbit/s]	**2.5**	**5**		**10**	
***pdMicrotick*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
adInternalRxDelay[samples] - **min$_N$**(*adInternalRxDelay$_N$*[samples])	0				
gdSampleClockPeriod[µs]	0.050	0.025		0.0125	
adPropagationDelayMin[Ch]^{Min}[µs]	0.349	0.175		0.087	
pDelayCompensation[Ch]^{Min}[µT]	**7**	**4**	**7**	**4**	**7**
adPropagationDelayMax[Ch]^{Max}[µs]	3.051	2.775		2.638	
pDelayCompensation[Ch]^{Max}[µT]	**61**	**55**	**111**	**105**	**211**

Table B-38: Calculations for *pDelayCompensation[Ch]*.

As a result, the parameter *pDelayCompensation[Ch]* shall be configurable between 4 and 211 µT.

[202] A starting point is given in chapter 7 of [Ung09] - if the exact propagation delays of all relevant sync frames are known it is possible to more accurately correct for the effects of the propagation delays. For example, by configuring the value of *pDelayCompensation* of a specific node to the smallest propagation delay affecting any sync frame that node receives, the precision formula is improved by replacing the absolute difference of largest and smallest propagation delay by one that maximizes over the local differences. A further optimization would attempt to evenly spread the measured offset term around zero, so the systematic offset drift is reduced and the NIT can be configured more efficiently.

B.4.27 pMacroInitialOffset

Consider the following assumption:

- *pMacroInitialOffset[Ch]* has to be in the range of *gdActionPointOffset* < *pMacroInitialOffset[Ch]* < *gdStaticSlot*.

Constraint 41:

pMacroInitialOffset[Ch][MT] = *gdActionPointOffset*[MT] + **ceil(** (*pDecodingCorrection*[µT] + *pDelayCompensation[Ch]*[µT]) / *aMicroPerMacroNom*[µT/MT] **)**

Bit Rate [Mbit/s]	**2.5**	**5**		**10**	
***pdMicrotick*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
gdActionPointOffset^{Min}[MT]	1	1		1	
gdActionPointOffset^{Max}[MT]	63	63		63	
pDecodingCorrection^{Min}[µT]	24	12	24	12	24
pDecodingCorrection^{Max}[µT]	56	40	80	68	136
pDelayCompensation[Ch]^{Min}[µT]	7	4	7	4	7
pDelayCompensation[Ch]^{Max}[µT]	61	55	111	105	211
aMicroPerMacroNom^{Min}[µT]	40	40	40	40	80
aMicroPerMacroNom^{Max}[µT]	120	120	240	240	240
pMacroInitialOffset[Ch]^{Min}[MT]	**2**	**2**	**2**	**2**	**2**
pMacroInitialOffset[Ch]^{Max}[MT]	**66**	**66**	**68**	**68**	**68**

Table B-39: Calculations for *pMacroInitialOffset[Ch]*.

As a result, the parameter *pMacroInitialOffset[Ch]* shall be configurable between 2 and 68 MT.

B.4.28 pMicroInitialOffset

Consider the following assumptions:

- The CSS process sets up a macrotick timing grid that is aligned to the primary TRP of the received frame used to start the macrotick generation process. The primary TRP is never actually determined, but it is assumed to occur before the detected secondary TRP as shown in Figure 3-11. Macrotick generation needs to be started at a point that represents an integral number of nominal macroticks from the assumed primary TRP. As a result, *pMicroInitialOffset[Ch]* is the number of microticks between the secondary TRP and the start of the subsequent macrotick on a macrotick grid aligned with the assumed primary TRP (see Figure B-1).

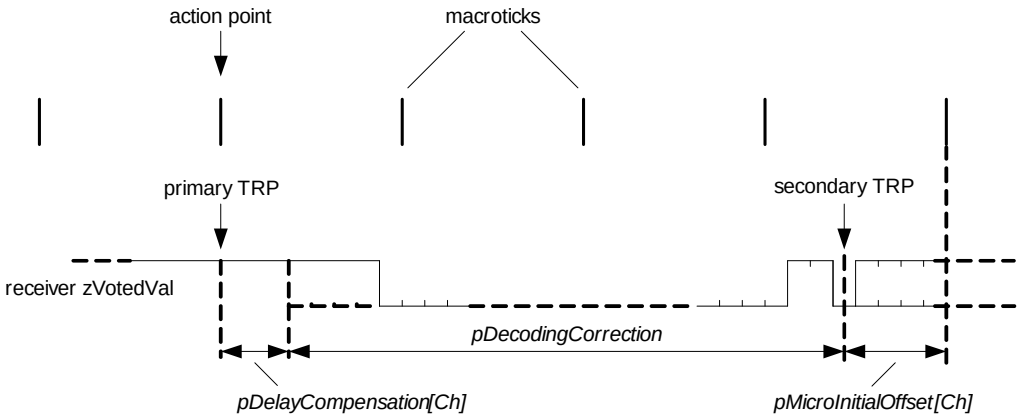

Figure B-1: Illustration of *pMicroInitialOffset[Ch]*.

Constraint 42:

pMicroInitialOffset[Ch][µT] = **floor(ceil(** (*pDecodingCorrection*[µT] +

pDelayCompensation[Ch][µT]) / *aMicroPerMacroNom*[µT/MT] **)** *

aMicroPerMacroNom[µT/MT] **)** -

(*pDecodingCorrection*[µT] + *pDelayCompensation[Ch]*[µT])

Since *aMicroPerMacroNom* must be in the range

cMicroPerMacroNomMin <= *aMicroPerMacroNom* <= *cMicroPerMacroNomMax*

the results of Constraint 42 will lie in the range from 0 to *cMicroPerMacroNomMax* - 1. As a result, the parameter *pMicroInitialOffset[Ch]* shall be configurable over a range of 0 to 239 µT.

Because of the mechanisms defined in the clock synchronization startup process the following constraint must also be fulfilled:

Constraint 43: [203]

pMicroInitialOffset[Ch][µT] < **floor(** (((5 + 2 * *gPayloadLengthStatic* + 3) * 10[gdBit] - *cdBSS*[gdBit]

+ 0.5 * *cdFES*[gdBit]) * *adBitMin*[µs/gdBit]) /

(*pdMicrotick*[µs/µT] / (1 - *gClockDeviationMax*)) **)**

B.4.29 pLatestTx

pLatestTx defines the last allowed starting point of a frame transmission in the dynamic segment. This parameter may be used by the system designer to ensure that if a node begins to transmit a frame in the dynamic segment that it will be allowed to complete the transmission (i.e., if transmission is allowed to start, there is enough time remaining in the dynamic segment to allow the transmission to complete). Obviously, the time required to complete the transmission is dependent on the size of the frame. Substituting the desired dynamic frame length *aPayloadLengthDynamic* for *aPayloadLength* in equation [21] results in

[36] *aFrameLengthDynamic* = *aFrameLength* with *aPayloadLength* = *pPayloadLengthDynMax*[204]

[203] Constraint 43 is necessary to ensure that events occur in the correct order in the clock synchronization startup process. For systems that follow the constraints in this appendix, this constraint will always be fulfilled and does not conflict with Constraint 42.

Using this, it is possible to develop an equation for a configuration of *pLatestTx* that would ensure that, once started, a transmission of a frame with a payload length less than or equal to *aPayloadLengthDynamic* will be able to be completed before the end of the dynamic segment.

Consider the following assumptions:

- After each frame the dynamic slot idle phase must be taken into account.
- The influence of the clock deviation (*gClockDeviationMax*) on the length of a macrotick must be taken into account.

With the definition of *aFrameLengthDynamic* in equation [36] and $adDTSLow^{Min}$ in equation [27], the constraint on *pLatestTx* is

Constraint 44:

pLatestTx[Minislot] <= **floor(** *gNumberOfMinislots*[Minislot] - (((*aFrameLengthDynamic*[gdBit] + $adDTSLow^{Min}$[gdBit]) * *adBitMax*[µs/gdBit] + *adMicrotickDistError*[µs]) / ((*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) * *gdMinislot*[MT/Minislot])) - *gdDynamicSlotIdlePhase*[Minislot] **)**

Constraint 44 defines, as a function of *aPayloadLengthDynamic*, the value that must be configured into *pLatestTx* to ensure that transmissions that begin will be allowed to complete.

A system designer can achieve various node behaviors through the choice of the value for *pLatestTx*. Some possibilities include:

- The node is not allowed to transmit in the dynamic segment. This could be achieved by setting *pLatestTx* to zero.
- The node is only allowed to transmit if all its possible frames could be completely transmitted. This could be achieved by setting *pLatestTx* to the value given by calculating Constraint 44 with *aPayloadLengthDynamic* set to the payload length of longest possible dynamic frame sent by the node.
- The node is only allowed to transmit if frames of a defined length could be completely transmitted. This length might be shorter than the longest possible frame sent by the node in the dynamic segment[205]. This could be achieved by setting *pLatestTx* to the value given by calculating Constraint 44 with *aPayloadLengthDynamic* set to the desired payload length.
- The node is only allowed to begin transmissions in the first portion of the dynamic segment. This could be achieved by setting *pLatestTx* to a value greater than zero, but still near the beginning of the dynamic segment.
- The node is allowed to start a transmission at any time in the dynamic segment even if the frame may not be completely transmitted (i.e., the *pLatestTx* mechanism is effectively circumvented). This could be achieved by setting *pLatestTx* to *gNumberOfMinislots*.

The value of *pLatestTx* must be configurable over a range of 0 to $gNumberOfMinislots^{Max}$ (see section B.4.17), and thus must be configurable over a range of 0 to 7988 minislots.

B.4.30 gdTSSTransmitter

Consider the following assumptions:

[204] This parameter may be different for each node. It is the length of the longest possible frame sent in the dynamic segment by the node under consideration.

[205] Such a configuration might be useful if, for example, a system designer knows that a node may send long frames, but only at the beginning of the dynamic segment.

- The first low phase of a frame (TSS), as seen by the receiver, may be changed at the beginning by an interval of up to $dFrameTSSLengthChange_{M,N}$ + $dFrameTSSEMIInfluence_{M,N}$ as the frame passes through the network. The amount of change depends on the particular bus drivers and active stars that are involved, and on the channel topology layout. The parameter *gdTSSTransmitter* must be chosen to be greater than the expected worst case truncation of a frame.
- Receiving nodes must receive at least one complete bit of the TSS phase.
- The nominal bit rates of different nodes in the cluster may differ by the maximum allowable clock deviation. The transmitted TSS duration must account for the case where the transmitter sends bits that are as short as allowed and the receiver expects bits that are as long as allowed.

Constraint 45: [206]

gdTSSTransmitter[gdBit] >= **ceil(** (*adBitMax*[µs] - **min$_{M,N}$(** $dFrameTSSLengthChange_{M,N}$[µs] + $dFrameTSSEMIInfluence_{M,N}$[µs] **)**) / *adBitMin*[µs/gdBit] **)**

Since every CAS or MTS symbol is prepended by a TSS, the resulting bit stream will be lengthened or shortened by an interval of up to $dSymbolLengthChange_{M,N}$ + $dSymbolEMIInfluence_{M,N}$ as the low phase passes through the network. The amount of length change depends on the active stars that are involved and on the channel topology layout.

In order to avoid shortening below the acceptance level for symbol reception the following Constraint must also be fulfilled:

Constraint 46:

gdTSSTransmitter[gdBit] >= **ceil(** (*cdCASRxLowMin*[gdBit] * *adBitMax*[µs/gdBit] - **min$_{M,N}$(** $dSymbolLengthChange_{M,N}$[µs] + $dSymbolEMIInfluence_{M,N}$[µs] **)**) / *adBitMin*[µs/gdBit] - *cdCAS*[gdBit] **)**

Bit Rate [Mbit/s]	2.5	5	10
adBitMax[µs]	0.4006	0.2003	0.10015
adBitMin[µs]	0.3994	0.1997	0.09985
$dFrameTSSLengthChange_{M,N}^{Min}$[µs][a]	-1.3		
$dFrameTSSEMIInfluence_{M,N}^{Min}$[µs][a]	-0.075		
$dSymbolLengthChange_{M,N}^{Min}$[µs][a]	-0.925		
$dSymbolEMIInfluence_{M,N}^{Min}$[µs][a]	-0.3		
$gdTSSTransmitter^{Max}$[gdBit]	**5**	**8**	**15**

Table B-40: Calculations for *gdTSSTransmitter*.

a. for a network with two active stars

By definition, the transmitted TSS must always consist of at least one bit. The calculation of the maximum number of bits required is shown in Table B-40. As a result, the parameter *gdTSSTransmitter* shall be configurable between 1 and 15 gdBit.

[206] A description of the $dFrameTSSLengthChange_{M,N}$ and $dFrameTSSEMIInfluence_{M,N}$ parameters may be found in [EPL10]. The maximum values of these parameters for systems with 0, 1, and 2 active stars may be found in [EPLAN10]. The configuration constraint is expressed in terms of the actual values of the TSS truncation present in the system under consideration. This may be less than the maximum values expressed in the [EPL10].

B.4.31 gdCASRxLowMax

Consider the following assumption:

- A CAS symbol shall be accepted even if it overlaps with another CAS symbol

The situations that can cause CAS overlaps during startup are quite complicated, and depend heavily on several characteristics of the system in question, including system topology, macrotick duration, propagation delays, ringing, symbol length change, etc. Further, the worst case situations themselves are also complicated, as some involve a combination of events that may be considered unlikely by some system designers (for example, the expiration of the startup noise timer sufficiently close to the completion of the reception of a CAS from another node), and vary from system to system. As a result of this complexity, presenting a single configuration constraint for all the possible combinations is beyond the scope of this specification.

In order to provide some guidance, the following conservative equations are presented.

Equation [37] gives an upper bound for situations in which there are no CAS collisions (i.e., where only a single node can act as a leading coldstarter).

[37] *gdCASRxLowMax*[gdBit] = **ceil(** ((*gdTSSTransmitter*[gdBit] + *cdCAS*[gdBit]) * *adBitMax*[µs/gdBit] + **max$_{M,N}$(** *dSymbolLengthChange*$_{M,N}$[µs] + *dSymbolEMIInfluence*$_{M,N}$[µs] + *anRingPath*$_{M,N}$ * *dRing*[µs] **)**) / *adBitMin*[µs/gdBit] **)**

Equation [38] gives an upper bound for situations in which there is an overlap of the CAS's of two coldstart nodes, assuming that each CAS experiences the maximum lengthening possible in the system

[38] *gdCASRxLowMax*[gdBit] = **ceil(** (2 * ((*gdTSSTransmitter*[gdBit] + *cdCAS*[gdBit]) * *adBitMax*[µs/gdBit] + **max$_{M,N}$(** *dSymbolLengthChange*$_{M,N}$[µs] + *dSymbolEMIInfluence*$_{M,N}$[µs] + *anRingPath*$_{M,N}$ * *dRing*[µs] **)**) + *adBitMin*[µs]) / *adBitMin*[µs/gdBit] **)**

Normally the minimum value for *gdCASRxLowMax* is limited by the minimum acceptance value *cdCASRxLowMin* which would imply a minimum value of 29 for *gdCASRxLowMax*. There are certain use cases, however, where the system designer might wish to eliminate the effects caused by a reception of a CAS during the startup procedure. This can be accomplished by configuring the value of *gdCASRxLowMax* to be less than *cdCASRxLowMin*[207]. In order to allow such configurations, the value of *gdCASRxLowMax* must be able to be configured to be less than *cdCASRxLowMin*.

In order to determine the maximum configurable value of *gdCASRxLowMax* a margin beyond the maximum value of equation [38] is taken into account. As a result, the parameter *gdCASRxLowMax* shall be configurable over a range of 28 to 254 gdBit.

B.4.32 gdWakeupTxIdle

The following constraint must be met:

[207] As the same detection is used for both CAS's and MTS's, such a system would also not be able to detect the reception of MTS's.

Constraint 47:

gdWakeupTxIdle[gdBit] = **ceil(** *cdWakeupTxIdle*[µs] / *gdBit*[µs/gdBit] **)**

Bit Rate [Mbit/s]	2.5	5	10
gdBit[µs]	0.4	0.2	0.1
gdWakeupTxIdle[gdBit]	**45**	**90**	**180**

Table B-41: Calculations for *gdWakeupTxIdle*.

As a result, the parameter *gdWakeupTxIdle* shall be configurable between 45 and 180 gdBit.

B.4.33 gdWakeupTxActive

The following constraint must be met:

Constraint 48:

gdWakeupTxActive[gdBit] = **ceil(** *cdWakeupTxActive*[µs] / *gdBit*[µs/gdBit] **)**

Bit Rate [Mbit/s]	2.5	5	10
gdBit[µs]	0.4	0.2	0.1
gdWakeupTxActive[gdBit]	**15**	**30**	**60**

Table B-42: Calculations for *gdWakeupTxActive*.

As a result, the parameter *gdWakeupTxActive* shall be configurable between 15 and 60 gdBit.

B.4.34 gdWakeupRxIdle

There are in principle two ways to configure the recognition of the idle or high phase that occurs between two low phases inside a WUS or WUDOP:

1. based on the length of the generated idle or high phase, a possible superposition of wakeup symbols and the maximum length change inside the network.
2. based on the acceptance criterion of the bus driver.

The following equation is based on the acceptance criterion of the bus driver ($dWU_{IdleDetect}$), under the assumption that the intended purpose behind the equation is to ensure that if a CC detects activity on the bus as a wakeup that the local bus driver would have been guaranteed to have detected the same activity as a wakeup.

[39] *gdWakeupRxIdle*[gdBit] = **ceil(** $dWU_{IdleDetect}^{Max}$[µs] / *adBitMin*[µs/gdBit] **)**

Note that this equation does not consider certain minor effects that could further shorten or lengthen the received idle or high phase. This is acceptable, since the acceptance criterion ($dWU_{IdleDetect}$) of the bus driver contains a substantial margin over the length of the idle or high phase generated by a FlexRay device that is transmitting WUP's or WUDOP's, even considering the effects of overlapping WUS's.

Because there are, however, multiple ways to configure this parameter, the required configuration range is not derivable directly from equation [39]. To allow different choices, the parameter *gdWakeupRxIdle* shall be configurable between 8 and 59 gdBit.

B.4.35 gdWakeupRxLow

There are in principle two ways to configure the recognition of the low phase of a received wakeup symbol:

1. based on the length of the transmitted low phase and the maximum shortening inside the network.
2. based on the acceptance criterion of the bus driver.

The following equation is based on the acceptance criterion of the bus driver ($dWU_{0Detect}$), under the assumption that the intended purpose behind the equation is to ensure that if a CC detects activity on the bus as a wakeup that the local bus driver would have been guaranteed to have detected the same activity as a wakeup.

[40] *gdWakeupRxLow*[gdBit] = **ceil(** $dWU_{0Detect}^{Max}$[µs] / *adBitMin*[µs/gdBit] **)**

Note that this equation does not consider certain minor effects that could further shorten or lengthen the received low phase. This is acceptable, since the acceptance criterion ($dWU_{0Detect}$) of the bus driver contains a substantial margin over the length of the low phase generated by a FlexRay device that is transmitting WUP's or WUDOP's.

Because there are, however, multiple ways to configure this parameter, the required configuration range is not derivable directly from equation [40]. To allow different choices, the parameter *gdWakeupRxLow* shall be configurable between 8 and 59 gdBit.

B.4.36 gdWakeupRxWindow

There are in principle two ways to configure the maximum window in which the phases necessary for wakeup detection must occur:

1. based on the maximum length of a pair of generated WUS's or
2. based on the acceptance criterion of the bus driver.

The following equation is based on the acceptance criterion of the bus driver ($dWU_{Timeout}$), under the assumption that the intended purpose behind the equation is to ensure that if a CC detects activity on the bus as a wakeup that the local bus driver would have been guaranteed to have detected the same activity as a wakeup.

[41] *gdWakeupRxWindow*[gdBit] = **floor(** $dWU_{Timeout}^{Min}$[µs] / *adBitMax*[µs/gdBit] **)**

Note that this equation does not consider certain effects that could further shorten or lengthen the various components required to detect a wakeup. This is acceptable, since the acceptance criterion ($dWU_{Timeout}$) of the bus driver contains a substantial margin over the length of the relevant components generated by a FlexRay device that is transmitting WUP's or WUDOP's.

Because there are, however, multiple ways to configure this parameter, the required configuration range is not derivable directly from equation [41]. To allow different choices, the parameter *gdWakeupRxWindow* shall be configurable between 76 and 485 gdBit.

B.4.37 gdIgnoreAfterTx

After the transmission of a WUS low phase a FlexRay CC normally switches immediately to reception in order to detect wakeup collisions. Due to digital and analog delays inside the CC and due to various delays inherent in the physical layer (e.g., TxEN - RxD loopback duration), the WUPDEC process will most likely observe some low bits that are the consequence of its own transmission instead of the transmission of another node. This effect is increased if there are also echoes or ringing on the physical layer of the transmitting node. A similar situation occurs after a frame transmission, delaying the detection of idle for the transmitter of a frame.

The parameter *gdIgnoreAfterTx* defines a phase in bit times after the transmission in which the bit strobing process of a transmitting node ignores the RxD input and instead assumes that the communication channel is idle.

The following constraint defines a value of *gdIgnoreAfterTx* that ensures that no bits would be strobed as low as a consequence of a transmission. This constraint is not strictly necessary, as a system may still operate properly even if one or more bits are strobed as low following the transmission. As a consequence, the following constraint should be considered to be optional - if this constraint is not met the system designer must ensure that the number of bits that could be strobed as low would be acceptable for their particular system.

Constraint 49:

gdIgnoreAfterTx[gdBit] >= **ceil(max$_M$(** ((*dCCTxEN01*[µs] + *adNodeTxRxai*$_M$[µs] + *dCCRxD01*[µs] + *dRxUncertainty*[µs]) / (*gdSampleClockPeriod*[µs/samples] / (1 + *gClockDeviationMax*)) + *adInternalRxDelay*$_M$[samples] + *cVotingDelay*[samples] - (*cStrobeOffset*[samples] - 1 samples)) **)** / *cSamplesPerBit*[samples/gdBit] **)**

with

adNodeTxRxai$_M$ = *dBDTxRxai* if node M is connected to the bus via a bus driver and

adNodeTxRxai$_M$ = *dStarTxRxai* if node M is connected to the bus via an active star-communication controller interface.

Table B-43 provides example calculations for *gdIgnoreAfterTx* under the assumption that the maximum echo/ring duration *dRxUncertainty* is no more than 0.250 µs.

Bit Rate [Mbit/s]	**2.5**	**5**	**10**
gdSampleClockPeriod[µs]	0.05	0.025	0.0125
dCCTxEN01^{Max}[µs]	0.025		
max(*dBDTxRxai*[µs]; *dStarTxRxai*[µs] **)**	0.325		
dCCRxD01^{Max}[µs]	0.01		
dRxUncertainty^{Max}[µs]	0.25		
max$_N$(*adInternalRxDelayN*^{Max}[samples] **)**	4		
gdIgnoreAfterTx[gdBit]	**2**	**4**	**7**

Table B-43: Calculations for *gdIgnoreAfterTx*.

Certain applications may experience ring/echo durations larger than 0.250 µs, requiring somewhat larger values for *gdIgnoreAfterTx*. In addition, some applications may never want the BITSTRB process to go into BLIND mode. As a result, the parameter *gdIgnoreAfterTx* shall be configurable over a range of 0 to 15 gdBit.

In addition to the previous constraint, in some systems it is possible that the selected value of *gdIgnoreAfterTx* might force a change to the constraints used for static slot size, minislot size, and symbol window size. This is due to the fact that following a transmission in a slot, minislot, or in the symbol window it is necessary that a node is actually able to receive communication activity that starts in the next slot, minislot, or segment (whichever is appropriate). In order to receive activity that is present it is necessary that the BITSTRB process is not currently operating in the BLIND mode, which is essentially equivalent to ensuring that the *gdIgnoreAfterTx* period is complete before the current slot, minislot, or symbol window ends. This creates the following additional constraints:

For the static slot size:

Constraint 50:

gdStaticSlot[MT] >= *gdActionPointOffset*[MT] + **ceil(** (*aFrameLengthStatic*[gdBit] + *gdIgnoreAfterTx*[gdBit] + 1 [gdBit]) * *adBitMax*[µs/gdBit] / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **)**

For the minislot size:

Constraint 51:

gdMinislot[MT/minislot] >= **ceil(** (*gdMinislotActionPointOffset*[MT] + **ceil(** (1 [gdBit] + *gdIgnoreAfterTx*[gdBit] + 1 [gdBit]) * *adBitMax*[µs/gdBit] / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **)**) / (1 [minislot] + *gdDynamicSlotIdlePhase*[minislot]) **)**

For a symbol window in which an MTS may be transmitted:

Constraint 52:

gdSymbolWindow[MT] >= *gdSymbolWindowActionPointOffset*[MT] + **ceil(** (*gdTSSTransmitter*[gdBit] + *cdCAS*[gdBit] + *gdIgnoreAfterTx*[gdBit] + 1 [gdBit]) * *adBitMax*[µs/gdBit] / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **)**

For a symbol window in which a WUDOP may be transmitted:

Constraint 53:

gdSymbolWindow[MT] >= *gdSymbolWindowActionPointOffset*[MT] + **ceil(** (5 * *gdWakeupTxActive*[gdBit] + 1 [gdBit] + *gdIgnoreAfterTx*[gdBit] + 1 [gdBit]) * *adBitMax*[µs/gdBit] / (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) **)**

Note while these constraints are always applicable, for the majority of practical systems these constraints will not be relevant as other constraints (in particular, those in sections B.4.11, B.4.12, and B.4.13) will require larger sizes. It is possible, however, that systems with low propagation delay and very good precision might be affected by Constraints 50 - 53.

Note that there is another potential situation that might result in a slightly different requirement on *gdIgnoreAfterTx*. During a wakeup process it is possible that two nodes begin their wakeup transmission at approximately the same point in time, and thus reach the end of the active low portion of the WUP at approximately the same time. The end of activity from each of the transmitters would propagate down the physical layer with a propagation delay characteristic of the physical layer. If both nodes are connected to the same linear bus (or to the same branch of an active star) it is possible in some circumstances for each node transmitting a wakeup to detect the activity of the other node, potentially causing both nodes to abort their wakeup transmission. This can only happen in systems where the propagation delay attributed to the physical layer is large compared to real time corresponding to *cdWakeupMaxCollision*, and where both nodes are connected to the same linear bus, or the same branch of an active star.

One possible mechanism to avoid this possibility would be to use a value for *gdIgnoreAfterTx* that is large enough to prevent a node from detecting the activity of another node that ends the active low phase of a WUP at the same instant as the local node. This could be done using a value for *gdIgnoreAfterTx* determined in a manner similar to Constraint 49 but also including the effects of the propagation delay associated with the physical layer between wakeup nodes that are connected to the same linear bus, or the same branch of an active star.

B.4.38 pKeySlotID

Nodes that do not have a key slot must set the value of *pKeySlotID* to zero.

Nodes that do have a key slot must configure *pKeySlotID* to a slot ID that lies within the static segment, and thus must satisfy the following constraint:

Constraint 54:

1 <= *pKeySlotID* <= *gNumberOfStaticSlots*

In addition, nodes for which *pKeySlotUsedForSync* is set to true must have a key slot, and thus the value of *pKeySlotID* for such nodes must satisfy Constraint 54.

The value of $gNumberOfStaticSlots^{Max}$ is *cStaticSlotIDMax*, and as a result the parameter *pKeySlotID* must be configurable over a range of 0 to *cStaticSlotIDMax*.

B.4.39 adTxMax

The electrical physical layer restricts the duration of any transmissions by a CC (see [EPL10]):

- The TxEN activation of a BD connected to a CC shall not exceed a duration of *dBDTxActiveMax*.
- Activity on an incoming branch of an active star shall not exceed a duration of *dBranchRx-ActiveMax*. This activity can be prolonged by ringing.

The parameter *adTxMax* defines the maximum transmission duration of a CC, i.e., the maximum allowable duration of TxEN activation of a CC for a specific system:

[42] *adTxMax*[µs] = $\mathbf{max_{M,N}}$**(if(** $nStarPath_{M,N}$ > 0**; min(** *dBDTxActiveMax*[µs]**;**

dBranchRxActiveMax[µs] - *dBDTxDM*[µs] -

if($anRingPath_{M,N}$ > 0**;** *dRing*[µs]**;** 0 **)** - ($nStarPath_{M,N}$ - 1) *

max((*dStarFES1LengthChange*[µs] + *dStarTSSLengthChange*[µs]); 0 **));**

dBDTxActiveMax[µs] **))**

B.4.40 gPayloadLengthStatic

Consider the following assumption:

- The duration of a static frame transmission shall not exceed the maximum transmission duration of a CC.

Therefore the following constraint must be met:

Constraint 55:

adTxStat[µs] <= *adTxMax*[µs]

with *adTxStat* being the upper bound for the static frame transmission duration, calculated by

[43] *adTxStat*[µs] = *aFrameLengthStatic*[gdBit] * *adBitMax*[µs/gdBit]

with the definition of *aFrameLengthStatic* as given in equation [22] in B.4.12.

Based on Constraint 55 and equation [43], the following constraint must be met:

Constraint 56:

gPayloadLengthStatic[two-byte word] <= **floor(** (*adTxMax*[µs] / *adBitMax*[µs/gdBit] -

gdTSSTransmitter[gdBit] - *cdFSS*[gdBit] - 80 [gdBit] - *cdFES*[gdBit]) /

20 [gdBit/two-byte word] **)**

Note that this constraint is automatically satisfied for systems operating at 5 or 10 Mbit/s.

By definition, *gPayloadLengthStatic* is configurable between 0 and *cPayloadLengthMax* two-byte words.

B.4.41 pPayloadLengthDynMax

Consider the following assumption:

- The duration of a dynamic frame transmission shall not exceed the maximum transmission duration of a CC.

Therefore the following constraint must be met:

Constraint 57:

adTxDyn[µs] <= *adTxMax*[µs]

with *adTxDyn* being the longest possible duration of a dynamic frame transmission[208], calculated by

[44] *adTxDyn*[µs] = **ceil(** ((*aFrameLength*[gdBit] + $adDTSLow^{Min}$[gdBit]) * *adBitMax*[µs/gdBit] + *adMicrotickDistError*[µs]) / ((*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) * *gdMinislot*[MT/Minislot]) **)** * ((*gdMacrotick*[µs/MT] / (1 - *gClockDeviationMax*)) * *gdMinislot*[MT/Minislot]) + *adBitMax*[µs] + *adMicrotickDistError*[µs]

with the definition of *aFrameLength* as given in equation [21] in B.4.12.

Note that this constraint is automatically satisfied for systems operating at 10 Mbit/s.

Constraint 57 applies to all dynamic frames, including frames with the longest payload of *pPayloadLength-DynMax*. As a result, this constraint indirectly places a constraint on the value of *pPayloadLengthDynMax*. A constraint for *pPayloadLengthDynMax* can't be derived directly from Constraint 57 due to the ceil-function in equation [44]. As a consequence, Constraint 57 has to be checked for any intended value of *pPayload-LengthDynMax*.

The equation below can be used to calculate a value for *pPayloadLengthDynMax* that is guaranteed to work for a given system, although it is possible that larger values for *pPayloadLengthDynMax* would also be acceptable (i.e., would satisfy Constraint 57).

[45] *pPayloadLengthDynMax*[two-byte word] <= **floor(** (((((*adTxMax*[µs] - *adBitMax*[µs] - *adMicrotickDistError*[µs]) / (*gdMacrotick*[µs/MT] / (1 - *gClockDeviationMax*))) - *gdMinislot*[MT]) * (*gdMacrotick*[µs/MT] / (1 + *gClockDeviationMax*)) - *adMicrotickDistError*[µs]) / *adBitMax*[µs/gdBit] - $adDTSLow^{Min}$[gdBit] - *gdTSSTransmitter*[gdBit] - *cdFSS*[gdBit] - 80 [gdBit] - *cdFES*[gdBit]) / 20 [gdBit/two-byte word] **)**

B.4.42 gCycleCountMax

Consider the following assumption:

- The clock synchronization algorithm and startup process requires that cycle counter values alternate between even and odd values. This includes the behavior of the cycle counter when it is updated following the cycle with *vCycleCounter* equal to *gCycleCountMax*. Since the cycle counter is reset to zero (an even value), the last cycle before the reset to zero must have an odd cycle counter value.

[208] This calculation represents an upper bound, but no transmission will actually reach this duration.

As a result, the following constraint must be fulfilled:

Constraint 58:

gCycleCountMax must be an odd integer.

B.5 Configuration of cluster synchronization method and node synchronization role

The node synchronization role and, indirectly, the cluster synchronization method is determined by several configuration parameters listed in Table B-44. Please note that the behavior of non-sync nodes is identical for all cluster synchronization methods.

Synchronization role	Synchronization method	*pKeySlotUsedForStartup*	*pKeySlotUsedForSync*	*pTwoKeySlotMode*	*pExternalSync*
coldstart node	TT-D cluster	true	true	false	false
coldstart node	TT-L cluster	true	true	true	false
coldstart node	TT-E cluster	true	true	true	true
sync node	TT-D cluster	false	true	false	false
non-sync node	TT-D, TT-L or TT-E cluster	false	false	false	false

Table B-44: Relationship of configuration parameters that determine the role of a FlexRay node.

Any combination not depicted in this table is illegal; thus the behavior of a node for any such combination is undefined.

If the optional parameter *pExternalSync* is not implemented, the node behaves as if it were set to false.

The parameter *pFallBackInternal* is only evaluated by a coldstart node in a TT-E cluster, i.e., a time gateway sink node. This parameter determines the behavior of the node in case that the synchronization to the time gateway source is lost or erroneous. If the parameter is set to true, the node will switch to its local clock and continue to operate in NORMAL_ACTIVE. If the parameter is set to false, the node will switch to *POC:halt*.

The parameter *pFallbackInternal* must be set to false if there is more than one coldstart node in a TT-E cluster.

B.6 Calculation of configuration parameters for nodes in a TT-L cluster

This section describes the configuration parameters that are only available for nodes operating in a TT-L cluster or that are calculated differently compared to operation in a TT-D cluster. For parameters not contained in this section please refer to section B.4.

B.6.1 gClusterDriftDamping

Consider the following assumption:

- The timing of a TT-L cluster is fully determined by the timing of the single TT-L coldstart node. As there is no feedback from the non-sync nodes of the TT-L cluster to the TT-L coldstart node, measurement errors cannot accumulate in the rate correction term. Thus it is unnecessary to compensate for this effect with the cluster drift damping term.

Constraint 59:

gClusterDriftDamping[µT] = 0 µT

B.6.2 TT-L cluster precision

A TT-L cluster contains only a single sync node and is therefore not capable of withstanding Byzantine faults. As a consequence, only a best-case precision can be given:

[46] *aBestCasePrecision*[µs] = 13 µT * *adMicrotickMax*[µs/µT] / (1 - *gClockDeviationMax*) +

adMicrotickMaxDistError[µs] + *adPropagationDelayMax*[µs] - *adPropagationDelayMin*[µs]

It should not be assumed that the precision of a cluster cannot exceed the term given in [46] in the presence of faults; it is just not possible to characterize the cluster behavior analytically.

It is further possible to compensate the propagation delay of the sync frames rather precisely by configuring the parameter *pDelayCompensation* in each node based on the actual propagation delay between the single coldstart node and the node in question; this would replace the propagation delay difference terms in equation [46] with the maximum over all nodes of the difference between the configured delay compensation term and the actual propagation delay between the sync node and the node in question. Such a configuration should only be done if the system designer has a good understanding of the actual propagation delays.

With the new definition of *aBestCasePrecision* given in equation [46] Constraint 5 remains valid.

In order for the values of the parameters that depend on the assumed precision to remain within their defined parameter ranges the value chosen for *aAssumedPrecision* in a TT-L cluster should lie within the range of allowable values of *aAssumedPrecision* for TT-D clusters as given in section B.4.2.5.

B.6.3 pSecondKeySlotID

Nodes that do not have a second key slot (*pTwoKeySlotMode* = false) must set the value of *pSecondKeySlotID* to zero.

Nodes that do have a second key slot (*pTwoKeySlotMode* = true) must configure *pSecondKeySlotID* to a slot ID that lies within the static segment, and thus must satisfy the following constraint

Constraint 60:

1 <= *pSecondKeySlotID* <= *gNumberOfStaticSlots*

Further, the second key slot must be different from the first key slot, thus

Constraint 61:

pKeySlotID != *pSecondKeySlotID*

B.6.4 gdActionPointOffset

Due to having only a single coldstart node, Constraint 13 is superfluous and must not be applied to TT-L clusters. Constraint 12 remains valid.

B.7 Calculation of configuration parameters for nodes in a TT-E cluster

This section describes the configuration parameters that are only applicable for nodes operating in a TT-E cluster or parameters that are calculated differently than for nodes operating in a TT-D cluster. Parameters not specifically mentioned in this section are configured in the same manner as for nodes operating in a TT-D cluster, i.e., are configured in the same manner as described in section B.4.

For some of the formulas of this section, the value of parameters of the time source cluster is needed. Thus it becomes necessary to distinguish between parameters of the time sink cluster and the time source cluster. To that effect, parameters of the time sink cluster stay unmarked, but parameters of the time source cluster are marked with a "source" suffix as in *adPropagationDelayMax*^{source}.

Further, some constraints affect only the coldstart nodes of a TT-E cluster. For such constraints, the configuration parameters are annotated with a "GWsink" suffix as in *pdMicrotick*^{GWsink}. Correspondingly configuration parameters of the time gateway source nodes are similarly marked, e.g., as *pdMicrotick*^{GWsource}.

B.7.1 gClusterDriftDamping

Consider the following assumption:

- The timing of a FlexRay cluster using the TT-E synchronization method is fully determined by the timing of the time source cluster. As there is no feedback from the time sink cluster to the time source cluster, the measurement errors of the nodes of the time sink clusters cannot accumulate in the rate correction term. Thus it is unnecessary to compensate for this effect with the cluster drift damping term.

Constraint 62:

gClusterDriftDamping[µT] = 0 µT

B.7.2 TT-E cluster precision

The precision of a TT-E cluster is not only dependent on the configuration and topology of the cluster in question, i.e., the time sink cluster, but also depends on the parameters and topology of the time source cluster.

In the following, the TT-E cluster precision values are given for an example system where the time source cluster is a TT-D cluster operating with worst-case precision and an example where the time source cluster is a TT-L cluster operating with best case precision. To determine the TT-E cluster precision for different configurations please refer to [Ung09].

The following term quantifies the effect of the difference in propagation delay between the time source and time sink clusters on the precision of the time sink cluster. In this term the index M runs over all the time gateway nodes of the time source cluster and the index N runs over all nodes of the time sink cluster.

[47] *aMixedTopologyError*[µs] = **max$_{M,N}$ (**

adPropagationDelayMax^{source}[µs] - *adPropagationDelayMin*[µs] -

pDelayCompensation$_M$^{source}[µT] * *pdMicrotick*$_M$^{source}[µs/µT] /

(1 + *gClockDeviationMax*^{source}) +

pDelayCompensation$_N$[µT] * *pdMicrotick*$_N$[µs/µT] / (1 - *gClockDeviationMax*) **;**

adPropagationDelayMax[µs] - *adPropagationDelayMin*^{source}[µs] -

pDelayCompensation$_N$[µT] * *pdMicrotick*$_N$[µs/µT] / (1 + *gClockDeviationMax*) +

pDelayCompensation$_M$^{source}[µT] * *pdMicrotick*$_M$^{source}[µs/µT] /

(1 - *gClockDeviationMax*^{source}) **)**

Bit Rate [Mbit/s]	2.5	5		10	
***pdMicrotick*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
***pdMicrotick*^{source}[µs]**	**0.050**	**0.050**	**0.050**	**0.025**	**0.025**
adPropagationDelayMax^{sourceMax}[µs]	3.051	2.775		2.638	
adPropagationDelayMin^{Min}[µs]	0.349	0.175		0.087	
pDelayCompensation$_M$^{sourceMin}[µT]	7	4	4	4	4
pDelayCompensation$_N$^{Max}[µT]	61	55	111	105	211
adPropagationDelayMax^{Max}[µs]	3.051	2.775		2.638	
adPropagationDelayMin^{sourceMin}[µs]	0.349	0.175		0.087	
pDelayCompensation$_N$^{Min}[µT]	7	4	7	4	7
pDelayCompensation$_M$^{sourceMax}[µT]	61	55	55	105	105
aMixedTopologyError^{Max}[µs]	**5.407**	**5.154**	**5.179**	**5.080**	**5.093**

Table B-45: Calculation of *aMixedTopologyError*.

Note that Table B-45 does not show all possible combinations of microtick durations in the time source and time sink clusters. The values shown in the table are those combinations that maximize the calculation of precision shown in Table B-46.

The following two sections supply bounds on the precision of the TT-E cluster for different types of time source clusters.

B.7.2.1 TT-E cluster precision for a TT-D worst-case precision time source cluster

The precision of a TT-E cluster with a TT-D or TT-L time source cluster subject to a worst-case precision can be calculated as:

[48] *aSinkPrecision*$^{TT\text{-}D\_source}$[µs] = (50 µT + 28 * *gClusterDriftDamping*^{source}[µT]) *
adMicrotickMax^{source}[µs/µT] / (1 - *gClockDeviationMax*^{source}) +
(13 µT + 3 * *gClusterDriftDamping*^{source}[µT]) *
adMicrotickMax [µs/µT] / (1 - *gClockDeviationMax*) +
adMicrotickMaxDistError[µs] + *aMixedTopologyError*[µs] +
2 * (*adPropagationDelayMax*^{source}[µs] - *adPropagationDelayMin*^{source}[µs])

Bit Rate [Mbit/s]	2.5	5		10	
***adMicrotickMax*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
***adMicrotickMax*^{source}[µs]**	**0.050**	**0.050**	**0.050**	**0.025**	**0.025**
gClusterDriftDamping^{sourceMax}[µT]	5				
aMixedTopologyError^{Max}[µs]	5.407	5.154	5.179	5.080	5.093
adPropagationDelayMax^{sourceMax}[µs]	3.051	2.775		2.638	
adPropagationDelayMin^{sourceMin}[µs]	0.349	0.175		0.087	
aSinkPrecision$^{TT\text{-}D\_sourceMax}$[µs]	**21.777**	**21.320**	**20.619**	**15.665**	**15.315**

Table B-46: Calculation of maximum precision in a sink cluster.

B.7.2.2 TT-E cluster precision for a TT-L time source cluster

The precision for TT-E cluster with TT-L time source cluster subject to a best-case precision can be calculated as:

[49] *aSinkPrecision*$^{TT\text{-}L\_source}$[µs] =

19 µT * *adMicrotickMax*^{source}[µs/µT] / (1 - *gClockDeviationMax*^{source}) +

13 µT * *adMicrotickMax*[µs/µT] / (1 - *gClockDeviationMax*) +

adMicrotickMaxDistError[µs] + *aMixedTopologyError*[µs] +

(*adPropagationDelayMax*^{source}[µs] - *adPropagationDelayMin*^{source}[µs])

Bit Rate [Mbit/s]	2.5	5		10	
***adMicrotickMax*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
***adMicrotickMax*^{source}[µs]**	**0.050**	**0.025**	**0.025**	**0.0125**	**0.0125**
aMixedTopologyError^{Min}[µs]	0				
min(*adPropagationDelayMax*^{source}[µs] - *adPropagationDelayMin*^{source}[µs] **)**	0				
aSinkPrecision$^{TT\text{-}L\_sourceMin}$[µs]	**1.652**	**1.177**	**0.826**	**0.588**	**0.413**

Table B-47: Calculation of the minimum precision in a sink cluster.

B.7.2.3 TT-E assumed precision

As a consequence of the different way of calculating the lower bound, Constraint 5 is replaced by

Constraint 63:

aAssumedPrecision[µs] >= *aSinkPrecision*[µs]

It is recommended that the assumed precision for the TT-E cluster be set to the value for *aSinkPrecision* for the appropriate source cluster type (i.e., according to B.7.2.1 if the time source cluster is a TT-D cluster, and

according to B.7.2.2 if the source cluster is a TT-L cluster). [Ung09] gives more detail about the relationship between the time source cluster and time sink cluster that can be used to improve the precision estimate but goes beyond the scope and intent of the specification.

Note that the calculations for *aSinkPrecision* make the assumption that if external clock correction is used in the time source cluster (i.e., *gExternOffsetCorrection* or *gExternRateCorrection* are non-zero for the time source cluster) that corresponding external correction terms are also applied in the non-sync nodes in the time sink cluster, and that the application of those external corrections take place at the same time as in the time source cluster.

Parameter ranges in this specification are calculated assuming that the maximum value of *aAssumedPrecision* is calculated using the worst case value for the assumed precision of a TT-D source cluster (see section B.4.2.5) and then applying Constraint 63 as an equality constraint. The configurable range of most parameters allows some amount of margin beyond this assumption. If a system designer of a TT-E cluster desires to use a value of *aAssumedPrecision* larger than the value of *aSinkPrecision* the resultant parameters must be carefully checked to ensure that they are still within the allowed parameter ranges.

B.7.3 pSecondKeySlotID

The configuration of *pSecondKeySlotID* for nodes operating in TT-E clusters is identical to the configuration of *pSecondKeySlotID* for nodes operating in TT-L clusters. Please refer to section B.6.3 and Constraint 60 and Constraint 61 for the configuration of this parameter.

B.7.4 Host-controlled external clock correction

In most cases the timing of a TT-E cluster is fully determined by the timing of the time source cluster and thus it is usually not necessary to make use of host-controlled external clock correction in a TT-E cluster. If, however, the time source cluster makes use of host-controlled external clock correction it may also be desirable for the time sink cluster to make a corresponding host-controlled external clock correction. If such a correction is used it should have the same value as the correction in the time source cluster, and should be applied in the same cycle as the correction in the time source cluster.

Constraint 64:

$$gExternOffsetCorrection[\mu s] = gExternOffsetCorrection^{source}[\mu s]$$

Constraint 65:

$$gExternRateCorrection[\mu s] = gExternRateCorrection^{source}[\mu s]$$

Although the non-sync nodes in the time sink cluster need to apply host-controlled external clock corrections in the indicated circumstances, the coldstart nodes for the TT-E cluster (i.e., the time gateway sink) should not perform any host-controlled external clock correction, as the clock correction from the time gateway source is imported into the time gateway sink by the CSP process (see Figure 8-21). As a result, the local external correction terms for the time gateway sink should be set to zero:

Constraint 66:

$$pExternOffsetCorrection^{GWsink}[\mu T] = 0\ \mu T$$

Constraint 67:

$$pExternRateCorrection^{GWsink}[\mu T] = 0\ \mu T$$

B.7.5 gdActionPointOffset

Due to being tightly coupled to the time source cluster, Constraint 13 is superfluous and must not be applied to TT-E clusters. Constraint 12 remains valid.

B.7.6 gMacroPerCycle

The constraints of section B.4.14 remain valid. In addition the following constraint must be observed.

Constraint 68:

gMacroPerCycle[MT] = *gMacroPerCycle*^{source}[MT]

This constraint puts implicit constraints on the configuration of the schedule of the time sink cluster.

B.7.7 gdMacrotick

The constraints of section B.4.4 remain valid. In addition the following constraint must be observed.

Constraint 69:

gdMacrotick[µs] = *gdMacrotick*^{source}[µs]

B.7.8 aOffsetCorrectionMax

Constraint 30 as well as equation [28] and [29] remain valid for TT-E systems. In order to compute an upper bound on the possible range of offset corrections the following equation should be used instead of equation [30]:

[50] *aOffsetCorrectionMax*[µs] <= **max(**

aNegativeOffsetCorrectionMax[µs] **;** *aPositiveOffsetCorrectionMax*[µs] **;**

aNegativeOffsetCorrectionMax^{source}[µs] **;** *aPositiveOffsetCorrectionMax*^{source}[µs] **)**

B.7.9 pOffsetCorrectionStart

The constraints and formulas of sections B.4.20 remain valid. Constraint 32 remains valid for non-sync nodes; for the time gateway sink node, it is replaced by

Constraint 70:

pOffsetCorrectionStart^{GWsink} = *pOffsetCorrectionStart*^{GWsource}

B.7.10 gdNIT

Consider the following assumption:

- The offset correction has to be applied within the NIT. To ensure this the following constraint has to be fulfilled.

Constraint 71:

gdNIT[MT] >= **max(** *adRemRateCalculation*[MT]**;** *adRemOffsetCalculation*[MT] +

adOffsetCorrection[MT]**;** *adRemOffsetCalculation*[MT] + *adOffsetCorrection*^{source}[MT] **)**

B.7.11 pdMicrotick

Consider the following assumption:

- The time gateway source node transfers its rate and offset correction values to the time gateway sink node. As the unit of these correction values is [µT], the time gateway sink node needs to use the same microtick length as the time gateway source node to correctly interpret the values.

The constraints of section B.4.4 remain valid. In addition the following constraint must be observed.

Constraint 72:

pdMicrotick^{GWsink}[µs] = *pdMicrotick*^{GWsource}[µs]

In addition to the above, in system topologies that have more than one time gateway there is a required relationship between the microtick durations of all of the time gateway sinks of a particular TT-E cluster. In particular, since the cycle offset between the time source and time sink cluster for each time gateway is defined as a number of microticks (*cdTSrcCycleOffset*, a protocol constant), and it is necessary that all time gateways in the same TT-E cluster use the same cycle offset, there is therefore a requirement that all time gateways in a TT-E cluster have identical microtick durations.

Constraint 73:

$\forall(i,j)$: $pdMicrotick^{GWsink\_i}$[µs] = $pdMicrotick^{GWsink\_j}$[µs]

B.7.12 adInitializationErrorMax

For the calculation of *adInitializationErrorMax* the Constraint 9 in section B.4.5 is valid. Due to the different precision in a TT-E cluster the ranges for *adInitializationErrorMax* are different as calculated in table B-18.

Bit Rate [Mbit/s]	**2.5**	**5**		**10**	
***adMicrotickMax*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
$aAssumedPrecision^{Min}$[µs] = $aSinkPrecision^{TT\text{-}L\_sourceMin}$[µs]	1.652	1.177	0.826	0.588	0.413
$gExternOffsetCorrection^{Min}$[µs]	0				
adPropagationDelayMax[µs] - *adPropagationDelayMin*[µs]	0				
$adInitializationErrorMax^{Min}$[µs]	**1.652**	**1.177**	**0.826**	**0.588**	**0.413**
$aAssumedPrecision^{Max}$[µs] = $aSinkPrecision^{TT\text{-}D\_sourceMax}$[µs]	21.777	21.320	20.619	15.665	15.315
$gExternOffsetCorrection^{Max}$[µs]	0.35				
$adPropagationDelayMin^{Min}$[µs]	0.349	0.175		0.087	
$adPropagationDelayMax^{Max}$[µs]	3.051	2.775		2.638	
$adInitializationErrorMax^{Max}$[µs]	**24.829**	**24.270**	**23.569**	**18.566**	**18.216**

Table B-48: Calculations for *adInitializationErrorMax*.

B.7.13 pdAcceptedStartupRange

For the calculation of *pdAcceptedStartupRange* Constraint 10 in section B.4.6 is valid. Due to the differences in the range of the precision the range of the *pdAcceptedStartupRange* parameter for a TT-E cluster is somewhat different than the range calculated in table B-19.

Bit Rate [Mbit/s]	2.5	5		10	
***adMicrotickMax*[µs]**	**0.050**	**0.050**	**0.025**	**0.025**	**0.0125**
aAssumedPrecision^{Min}[µs] = *aSinkPrecision*$^{TT\text{-}L\_source Min}$[µs]	1.652	1.177	0.826	0.588	0.413
adInitializationErrorMax^{Min}[µs]	1.652	1.177	0.826	0.588	0.413
pdMicrotick^{Max}[µs]	0.050	0.050	0.025	0.025	0.0125
pdAcceptedStartupRange^{Min}[µT]	**67**	**48**	**67**	**48**	**67**
aAssumedPrecision^{Max}[µs] = *aSinkPrecision*$^{TT\text{-}D\_source Max}$[µs]	21.777	21.320	20.619	15.665	15.315
adInitializationErrorMax^{Max}[µs]	24.829	24.270	23.569	18.566	18.216
pdMicrotick^{Min}[µs]	0.050	0.025	0.025	0.0125	0.0125
pdAcceptedStartupRange^{Max}[µT]	**934**	**1827**	**1771**	**2743**	**2687**

Table B-49: Calculations for *pdAcceptedStartupRange*.

The lower bound of *pdAcceptedStartupRange* is given by the calculation in a TT-D cluster (see section B.4.6, table B-19) and the upper bound by the calculation in a TT-E cluster (table B-49). As a result, the parameter *pdAcceptedStartupRange* must be configurable over a range of 29 to 2743 µT.

B.7.14 gCycleCountMax

Consider the following assumption:

- *gCycleCountMax* in the time sink cluster should be identically configured like *gCycleCountMax* in the time source cluster.

Constraint 74:

gCycleCountMax = *gCycleCountMax*^{source}

Appendix C
Wakeup Application Notes

This appendix contains some application notes related to the coordination of the host microcontroller, the FlexRay communication controller, and the bus drivers during the wakeup process as well as describing some techniques that can be used to perform wakeup during operation. This appendix is not intended to be complete, but merely to provide some example strategies that touch on some of the issues that will be faced by a system designer with respect to wakeup coordination.

Note the control and indication behavior of the bus driver is fairly complex. For example, certain indications are only available in certain BD operating modes. The descriptions in this appendix are at a high level, i.e., they merely indicate what needs to be done without explicitly indicating the detailed BD commands, modes, indications, etc. necessary to do it. Refer to [EPL10] for additional details.

C.1 Wakeup initiation by the host

A host that wants to initiate a wakeup of the cluster should first check its bus driver(s) to see if they have received wakeup patterns. If the bus driver of a channel did not receive a wakeup pattern, and if there is no startup or communication in progress, the host shall try to wake this channel[209].

The host should not wake channels whose bus drivers have received a wakeup pattern unless additional information indicates that startup is not possible without an additional wakeup of those channels.[210]

A single-channel node in a dual-channel cluster can trigger a cluster wakeup by waking its attached channel. This wakes up all nodes attached to this channel, including the coldstart nodes, which are always dual-channel. Any coldstart node that deems a system startup necessary will then wake the remaining channel before initiating communication startup.

C.1.1 Single-channel nodes

This section describes the wakeup behavior of single-channel nodes in single- or dual-channel clusters. The bus driver is assumed to be in the *BD_Sleep* or *BD_Standby* mode. The host is assumed to have determined that a cluster wakeup should be triggered.

1. The host first configures the communication controller.
2. The host checks whether a wakeup pattern was received by the bus driver.
3. The host puts the bus driver into the *BD_Normal* mode.
4. If a wakeup pattern was received by the bus driver, the node should enter the startup (step 8) instead of performing a wakeup. If no wakeup pattern was received by the bus driver, the node may perform a wakeup of the attached channel (which will eventually wake both channels of a dual-channel cluster).
5. The host configures *pWakeupChannel* to the attached channel.
6. The host commands the communication controller to begin the wakeup procedure.
7. After its wakeup attempt is complete the communication controller returns the result of the wakeup attempt.

[209] The host must distinguish between a local wakeup event and a remote wakeup received via the channel. This information is accessible at the bus driver.

[210] This is done to speed up the wakeup process and to limit the amount of traffic on the channels, which reduces the number of collisions during this phase.

8. The host commands the communication controller to commence startup and, possibly after a delay (see section C.4), to leave the coldstart inhibit mode.

C.1.2 Dual-channel nodes

This section describes the wakeup behavior of dual-channel nodes in dual-channel clusters.

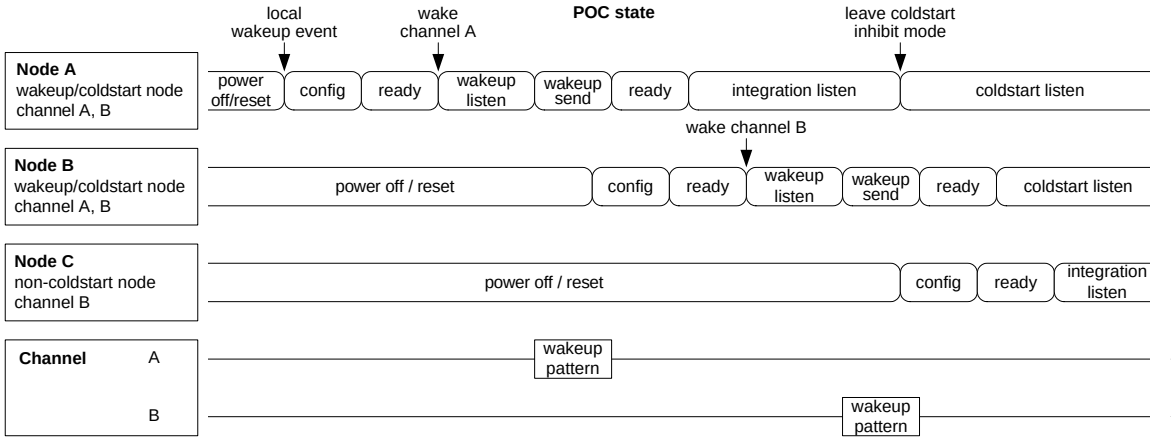

Figure C-1: A short example of how the wakeup of two channels can be accomplished in a fault-tolerant way by coldstart nodes.[211]

A communication controller is not allowed to send a wakeup pattern on both channels at the same time. If it is necessary to wake both channels, the host can only wake them one at a time.

To avoid certain types of failures, a single communication controller should not wake up both channels. Instead, a different controller should wake up each channel.

To accomplish this, a communication controller that has received a local wakeup event proceeds normally and only wakes a single channel, e.g., channel A (see Figure C-1). The completion of the wakeup causes the node to enter the *POC:ready* state, and also causes the node to automatically enter the coldstart inhibit mode. Following this, the host does not wake the other channel but rather enters startup. The host should keep the node in the coldstart inhibit mode until it detects a wakeup pattern on channel B. Once a wakeup is detected, the host should issue the ALLOW_COLDSTART command, allowing the node to actively coldstart the cluster.

Two example wakeup strategies are now given as examples to demonstrate how cluster wakeup can be accomplished. These strategies are not concerned with the details of error recovery and therefore do not show error handling mechanisms for several error situations that could occur.

C.1.2.1 Wakeup pattern reception by the bus driver

The bus drivers are assumed to be in the *BD_Sleep* or *BD_Standby* mode. The host is assumed to have determined that a cluster wakeup should be triggered.

1. The host first configures the communication controller. It assumes both channels to be asleep.
2. The host checks which of the bus drivers has received a wakeup pattern.
3. The host puts all bus drivers that have received a wakeup pattern into *BD_Normal* mode (these channels can be assumed to be awake).

[211] Note that there is no requirement that a wakeup node be a coldstart node, or that a coldstart node be a wakeup node. In this example the wakeup nodes are also coldstart nodes, but this is not required.

4. If both channels are awake, the host can proceed to startup (step 10).
   If both channels are asleep, the host shall wake up one of them.
   If one channel is asleep and one channel is awake, a non-coldstart host may wake up the channel that is asleep, but a coldstart host must wake up the channel that is asleep.
5. The host configures *pWakeupChannel* to the channel to be awakened.
6. The host puts the bus driver of *pWakeupChannel* into *BD_Normal* mode.
7. The host commands the communication controller to begin the wakeup procedure.
8. The communication controller returns the result of the wakeup attempt.
9. If the result of the wakeup attempt is TRANSMITTED, the host assumes *pWakeupChannel* to be awake and proceeds to startup (step 10).
   If the result of the wakeup attempt is RECEIVED_HEADER or COLLISION_HEADER, the host can assume that both channels are awake. It puts any remaining sleeping bus driver into *BD_Normal* mode and proceeds to startup (step 10).
   If the result of the wakeup attempt is RECEIVED_WUP or COLLISION_WUP, the host assumes *pWakeupChannel* to be awake (return to step 4).
   If the result of the wakeup attempt is COLLISION_UNKNOWN, an application-specific recovery strategy has to be employed, which is not covered by this document.
10. The host commands the communication controller to begin the startup procedure.
11. If all channels are awake, the host may immediately command the communication controller to leave the coldstart inhibit mode by issuing an ALLOW_COLDSTART command. Otherwise, the host should leave the CC in coldstart inhibit mode and wait until the bus driver of the still sleeping channel signals the reception of a wakeup pattern. This bus driver shall then be put into the *BD_Normal* mode. As soon as all attached channels are awake, the host may command the communication controller to leave the coldstart inhibit mode.

This method has the disadvantage that the channel that is not *pWakeupChannel* cannot be listened to during the *POC:wakeup listen* state. If the bus driver of *pWakeupChannel* channel is subject to an incoming link failure, ongoing communication might be disturbed.

C.1.2.2 Wakeup pattern reception by the communication controller

The wakeup pattern receiver of the communication controller is active as long as the CODEC is in READY or NORMAL mode. During this time the reception of a wakeup pattern will be signaled to the FSP and from there to the CHI.

The bus drivers are assumed to be in the *BD_Sleep* or *BD_Standby* modes. The host is assumed to have determined that a cluster wakeup should be triggered.

1. The host first configures the communication controller. It assumes both channels to be asleep.
2. The host checks which of the bus drivers has received a wakeup pattern.
3. The host puts both bus drivers into *BD_Normal* mode.
4. If both channels are awake, the host can proceed to startup (step 10).
   If both channels are asleep, the host shall awake one of them.
   If one channel is asleep and one channel is awake, a non-coldstart host may wake up the channel that is asleep, but a coldstart host must wake up the channel that is asleep.
5. The host configures *pWakeupChannel* to the channel that to be awakened.
6. The host commands the communication controller to begin the wakeup procedure.
7. The communication controller returns the result of the wakeup attempt.
8. If the result of the wakeup attempt is TRANSMITTED, the host assumes *pWakeupChannel* to be awake and proceeds to startup (step 10).
   If the result of the wakeup attempt is RECEIVED_HEADER or COLLISION_HEADER, the host assumes all attached channels to be awake and proceeds to startup (step 10).

If the result of the wakeup attempt is RECEIVED_WUP or COLLISION_WUP, the host assumes *pWakeupChannel* to be awake (return to step 4).
If the result of the wakeup attempt is COLLISION_UNKNOWN, an application-specific recovery strategy has to be employed, which is not covered here.

9. The host commands the communication controller to begin the startup procedure.
10. If all channels are awake, the host may immediately command the communication controller to leave the coldstart inhibit mode by issuing an ALLOW_COLDSTART command. Otherwise, the host should leave the CC in the coldstart inhibit mode and wait until the wakeup pattern detector of the communication controller detects a wakeup pattern on the channel that is assumed to be asleep (this could already have occurred during one of the former steps). Once a wakeup pattern is detected the channel is assumed to be awake. As soon as all attached channels are awake, the host may command the communication controller to leave the coldstart inhibit mode.

C.2 Host reactions to status flags signaled by the communication controller

This section defines the status information that the communication controller can return to the host as result of its wakeup attempt and the recommended reaction of the host.

C.2.1 Frame header reception without decoding error

When a frame header without decoding error is received by the communication controller on either available channel while in the *POC:wakeup listen* (or *POC:wakeup detect*) state the communication controller aborts the wakeup, even if channel *pWakeupChannel* is still silent.

The host shall not command the communication controller to initiate additional wakeup attempts, since this could disturb ongoing communication. Instead, it shall command the communication controller to enter the startup to integrate into the apparently established cluster communication.

C.2.2 Wakeup pattern reception

The communication controller has received a wakeup pattern on channel *pWakeupChannel* while in the *POC:wakeup listen* (or *POC:wakeup detect*) state. This indicates that another node is already waking up this channel. To prevent collisions of wakeup patterns on channel *pWakeupChannel*, the communication controller aborts the wakeup.

If another channel is available that is not already awake, the host must determine whether the communication controller is to wake up this channel. If all available channels are awake, the host shall command the communication controller to enter startup.

C.2.3 Wakeup pattern transmission

The communication controller has transmitted the complete wakeup pattern on channel *pWakeupChannel*. The node can now proceed to startup.

C.2.4 Termination due to unsuccessful wakeup pattern transmission

The communication controller was not able to transmit a complete wakeup pattern because its attempt to transmit it resulted in at least *cdWakeupMaxCollision* occurrences of continuous logical LOW during the idle phase of a wakeup pattern. Possible reasons for this are heavy EMI disturbances on the bus or an internal error.[212]

[212] The collision of a wakeup pattern transmitted by this node with another wakeup pattern generated by a fault-free node will generally not result in this exit condition. Such a collision can be recognized after entering the *POC:wakeup detect* state and would be signaled by setting the variable *vPOC!WakeupStatus* to COLLISION_WUP.

Since no complete wakeup pattern has been transmitted, it cannot be assumed that all nodes have received a wakeup pattern. The host may use the retransmission procedure described in section C.3.

C.3 Retransmission of wakeup patterns

Some events or conditions may prevent a cluster from waking up even though a wakeup attempt has been made (possibly even without the transmitting communication controller being able to immediately detect this failure[213]). The host detects such an error when the cluster does not start up successfully after the wakeup.

The host may then initiate a retransmission of the wakeup pattern. The procedure described in section 7.1.2 shall be used to transmit a wakeup pattern on channel *pWakeupChannel*.

Note that this might disturb ongoing communication of other nodes if the node initiating the wakeup procedure is subject to an incoming link failure or a fault in the communication controller. The host must ensure that such a failure condition will not lead to a permanent disturbance on the bus.

C.4 Transition to startup

It cannot be assumed that all nodes and stars need the same amount of time to become completely awake and to be configured.

Since at least two nodes are necessary to start up the cluster communication, it is advisable to delay any potential startup attempt of the node having initiated the wakeup by the minimal amount of time it takes another coldstart node to become awake, to be configured, and to enter startup[214]. Otherwise, the wakeup-initiating coldstart node may fail in its startup attempt with an error condition that is not distinguishable from a defective outgoing link (the communication controller reports no communication partners; see section C.3).

The coldstart inhibit mode can be used to effectively deal with this situation. A communication controller in this mode will only participate in the startup attempts made by other nodes but not initiate one itself. The coldstart inhibit mode is set automatically prior to entry to the *POC:ready* state; the host should not cause the CC to exit this mode (by issuing an ALLOW_COLDSTART command) before the above mentioned minimal time for another node to become ready for the communication startup. However, the host shall issue the ALLOW_COLDSTART command as soon as all coldstart nodes are awake in the fault-free case[215]. Please refer to section 7.2.3 for further details of this mode.

C.5 Wakeup during operation

This section describes wakeup during operation methods that a communication controller can use during the operation of a cluster to trigger the wakeup of other nodes with BD's in the sleep or standby mode when those BD's support the optional remote wakeup event detection capability described in [EPL10].

C.5.1 Principles

During the operation of a FlexRay network it is possible that some nodes do not take part in the normal FlexRay wakeup procedure (e.g. when they power up too late) or have for some reason gone to standby or sleep after they already were part of an ongoing communication (e. g. because of the reset of a node).

[213] E.g. an erroneous star that needs significantly more time to start up and to be able to forward messages.
[214] This parameter depends heavily on implementation details of the components used and the ECU structure.
[215] If these times are not known at system design time, it is advised to exit the coldstart inhibit mode late rather than early.

The normal wakeup procedure described in section 7.1 would require a node attempting to wake up other nodes be removed from the normal communication, disrupting the transmission and reception capability of that node. In addition, such a wakeup attempt would be unsuccessful because the ongoing communication in the cluster would cause the abortion of the wakeup procedure before wakeup patterns would be sent. In such cases it is necessary that the nodes already in operation be capable of causing other nodes to wake up without requiring the entire cluster to be shutdown and without disrupting their own communications.

The protocol offers two methods to provide a remote wakeup during normal operation capability:

1. frame-based transmission of a sequence of bus activity
2. pattern-based transmission of a pattern

The first method is completely transparent to the protocol operation and is therefore outlined in this section. The second method is explicitly supported by the mechanisms defined for the FlexRay protocol.

C.5.1.1 Frame-based wakeup during operation

[EPL10] defines the payload of a FlexRay frame that, when transmitted at a data rate of 10 Mbit/s, is guaranteed to cause a BD in the sleep or standby mode to detect a remote wakeup event. A node can cause a wakeup during normal operation by transmitting a frame with this special payload. This can be done in either the static or the dynamic segment, but in the static segment the entire payload must fit within the payload length of static frames defined by *gPayloadLengthStatic*. A node whose BD is in the standby or sleep mode that receives such a frame will signal the reception of a remote wakeup event to the host, and then operation can proceed as defined in the other sections of this chapter.

Note that the frame payload defined in [EPL10] is only guaranteed to cause a wakeup for systems operating at 10 Mbit/s. At bit rates of 5 and 2.5 Mbit/s the alternating data pattern of the BSS can interfere with the remote wakeup detection process in the BD's and prevent detection of wakeup attempts based on frame payloads.

During normal operation, the reception of a frame-based wakeup would not be detected by the protocol's wakeup pattern decoding process and thus will not be signaled to the host. The reception of a frame and its payload will, however, be signaled to the host by the normal reception mechanisms, and this allows some possibility to verify that other nodes are transmitting frame-based wakeups as expected.

C.5.1.2 Pattern-based wakeup during operation

The FlexRay protocol supports the transmission of a dedicated wakeup during operation pattern (WUDOP) within the symbol window. The WUDOP is described in section 3.2.1.2.3 and will cause the detection of a remote wakeup by BD's regardless of the bit rate of the cluster (i.e., unlike frame-based methods, the pattern-based mechanism will work in systems using bit rates of 5 and 2.5 Mbit/s).

The host has control over the transmission of the WUDOP by means of the mechanisms defined in section 9.3.1.2.2. The WUDOP is not collision resilient - the system designer must ensure that no more than one node transmits a WUDOP in any given instance of the symbol window.

A node whose BD is in the standby or sleep mode that receives a WUDOP will signal the reception of a remote wakeup event to the host, and then operation can proceed as defined in the other sections of this chapter.

During normal operation, the reception of a WUDOP in the symbol window or NIT would be detected by the protocol's wakeup pattern decoding process and signaled to the host (see section 9.3.1.3.2). This allows a node in operation to verify that other nodes are transmitting WUDOPs as expected.