FlexRay Communications System

Protocol Specification

Version 2.1

Revision A

Disclaimer

This specification as released by the FlexRay Consortium is intended **for the purpose of information only**. The use of material contained in this specification requires membership within the FlexRay Consortium or an agreement with the FlexRay Consortium. The FlexRay Consortium will not be liable for any unauthorized use of this Specification.

Following the completion of the development of the FlexRay Communications System Specifications commercial exploitation licenses will be made available to End Users by way of an End User's License Agreement. Such licenses shall be contingent upon End Users granting reciprocal licenses to all Core Partners and non-assertions in favor of all Premium Associate Members, Associate Members and Development Members.

All details and mechanisms concerning the bus guardian concept are defined in the FlexRay Bus Guardian Specifications.

The FlexRay Communications System is currently specified for a baud rate of 10 Mbit/s. It may be extended to additional baud rates.

The Core Partners of the FlexRay Consortium are BMW AG, DaimlerChrysler AG, Freescale Halbleiter Deutschland GmbH, General Motors Corporation, Philips GmbH, Robert Bosch GmbH, and Volkswagen AG.

Table of Contents

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, multi-master clock synchronization[1], error detection and signaling, error containment on the physical layer through the use of a bus guardian device, and scalable fault tolerance[2].

1.2 References

1.2.1 FlexRay consortium documents

[EPL05] FlexRay Communications System - Electrical Physical Layer Specification, v2.1 Revision A, FlexRay Consortium, December 2005.

[EPLAN05] FlexRay Communications System - Electrical Physical Layer Application Notes, v2.1 Revision A, FlexRay Consortium, December 2005.

[DLLCT05] FlexRay Communications System - Data Link Layer Conformance Test Specification, v1.0, FlexRay Consortium, December 2005.

[Req05] FlexRay Requirements Specification, v2.100, FlexRay Consortium, December 2005.

[Mül01] B. Müller, “On FlexRay Clock Synchronisation”, Robert Bosch Corporation, 2001 (internal document).

[Ung02] J. Ungermann, “Performance of the FlexRay Clock Synchronisation”, Philips GmbH, 2002 (internal document).

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.

[Pet72] W. W. Peterson and E. J. Weldon, Jr., Error-Correcting Codes, 2nd ed., Cambridge MA: M.I.T. Press, 1972.

[Rau02] M. Rausch, "Optimierte Mechanismen und Algorithmen in FlexRay", Elektronik Automotive, pp. 36-40, December 2002.

[1] Multi-master clock synchronization refers to a synchronization that is based on the clocks of several (three 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 or without bus guardians, clusters with many or few sync nodes, etc.).

[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.

1.3 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

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.

bus guardian

an electronic component that protects a channel from interference caused by communication that is not temporally aligned with the cluster's communication schedule by limiting the times that a communication controller can transmit to those times allowed by the schedule.

channel

see communication channel.

channel idle

the condition of medium idle as perceived by each individual node in the network.

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) or by star couplers (star topology).

coldstart node

a node capable of initiating the communication startup procedure on the cluster by sending startup frames.

communication channel

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

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 counter

the number of the current communication cycle.

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 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.

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.

medium idle

the condition of the physical transmission medium when no node is actively transmitting on the physical transmission medium.

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.

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 electrical or optical 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.

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.

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.

startup slot

communication slot in which a startup frame is sent.

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 slot

communication slot in which a sync frame is sent.

1.5 Acronyms and abbreviations

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
EMC	Electromagnetic Compatibility
ERRN	Error Not signal
FES	Frame End Sequence
FSP	Frame and Symbol Processing
FSS	Frame Start Sequence
FTDMA	Flexible Time Division Multiple Access (media access method)
FTM	Fault Tolerant Midpoint
ID	Identifier
INH	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
RxEN	Receive data enable signal from bus driver
SDL	Specification and Description Language
SPI	Serial Peripheral Interface
STBN	Standby Not signal
SuF	Startup Frame
SW	Symbol Window
SyF	Sync Frame
TDMA	Time Division Multiple Access (media access method)
TRP	Time Reference Point
TSS	Transmission Start Sequence

TxD	Transmit Data signal from CC
TxEN	Transmit Data Enable Not signal from CC
WUP	Wakeup Pattern
WUS	Wakeup Symbol
WUPDEC	Wakeup Pattern Decoding Process

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 Parameter	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 *brown 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.

Terms which are important for FlexRay are highlighted in *red italics* at their first (or defining) instance in the text. An example is the term *communication element*.

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.

The SDL in this specification describes a dual channel FlexRay device. As a result, there are a number of mechanisms that exhibit behavior that is specific to a given channel. It is also possible to have FlexRay devices that only support a single channel, or that may be configured to support a single channel. Implementers of single channel devices will have to make appropriate adjustments to eliminate the effects of mechanisms and signals that do not exist in devices operating in a single channel mode.

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.

[3] If the duration time expression is zero or negative then the timer is started and expires immediately.

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, 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*).

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.

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 process 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.

1.8 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 [EPL05].

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[5].

[4] This is in contrast to the *vMicrotick* variable, which is reset to zero at the beginning of each cycle.
[5] For example, it is not allowed for a communication controller to connect to Channel A of one cluster and Channel B of another cluster.

1.8.1 Passive bus topology

Figure 1-1 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).

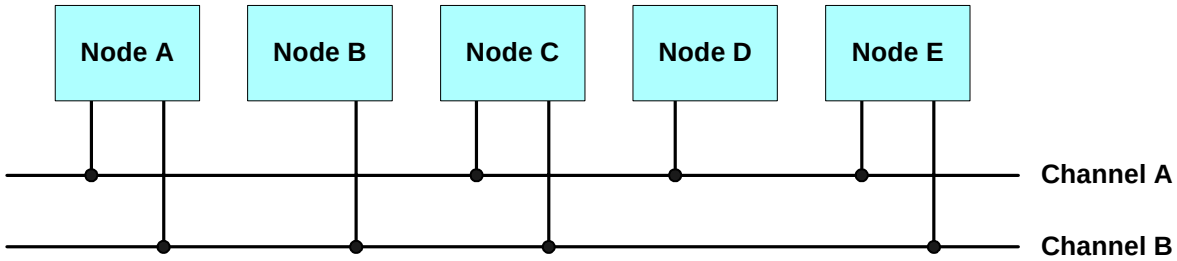

Figure 1-1: Dual channel bus configuration.

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

The number of nodes connected to a bus can vary from 2 up to *nStub* nodes (see [EPL05] for details).

1.8.2 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 star couplers on a network channel[6]. The incoming signal received by the star coupler is actively driven to all communication nodes[7].

The configuration of a single redundant star network is shown in Figure 1-2. The logical structure (i.e., the node connectivity) of this topology is identical with that shown in Figure 1-1. It is also possible to create a single, non-redundant star topology that has the same logical structure as the single bus mentioned above.

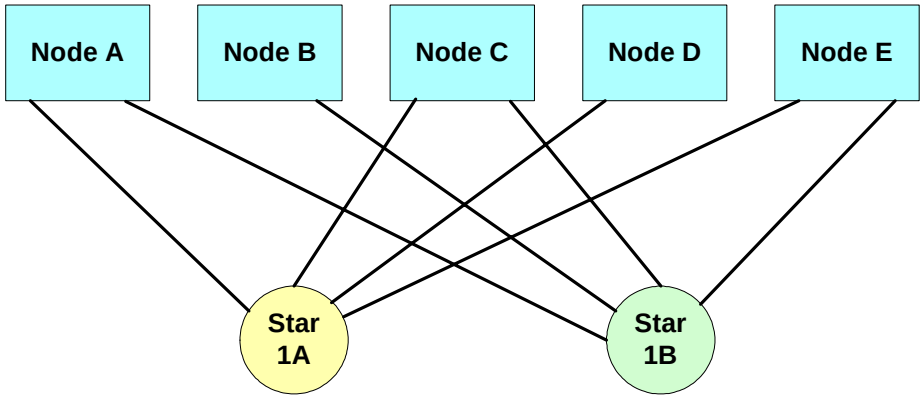

Figure 1-2: Dual channel single star configuration.

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

[7] With the obvious exception of the node transmitting the original signal

Figure 1-3 shows a single channel network built with two star couplers. Each node has a point-to-point-connection to one of the two star couplers. The first star coupler (1A) is directly connected to the second star coupler (2A).

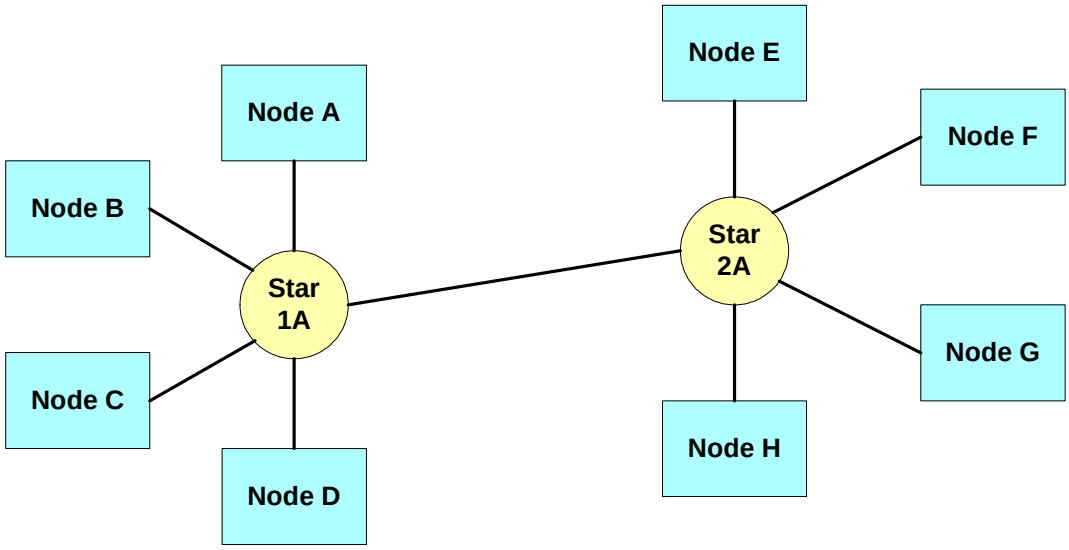

Figure 1-3: Single channel cascaded star configuration.

Note that it is also possible to have a redundant channel configuration with cascaded stars. An example of such a configuration is Figure 1-4. 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.

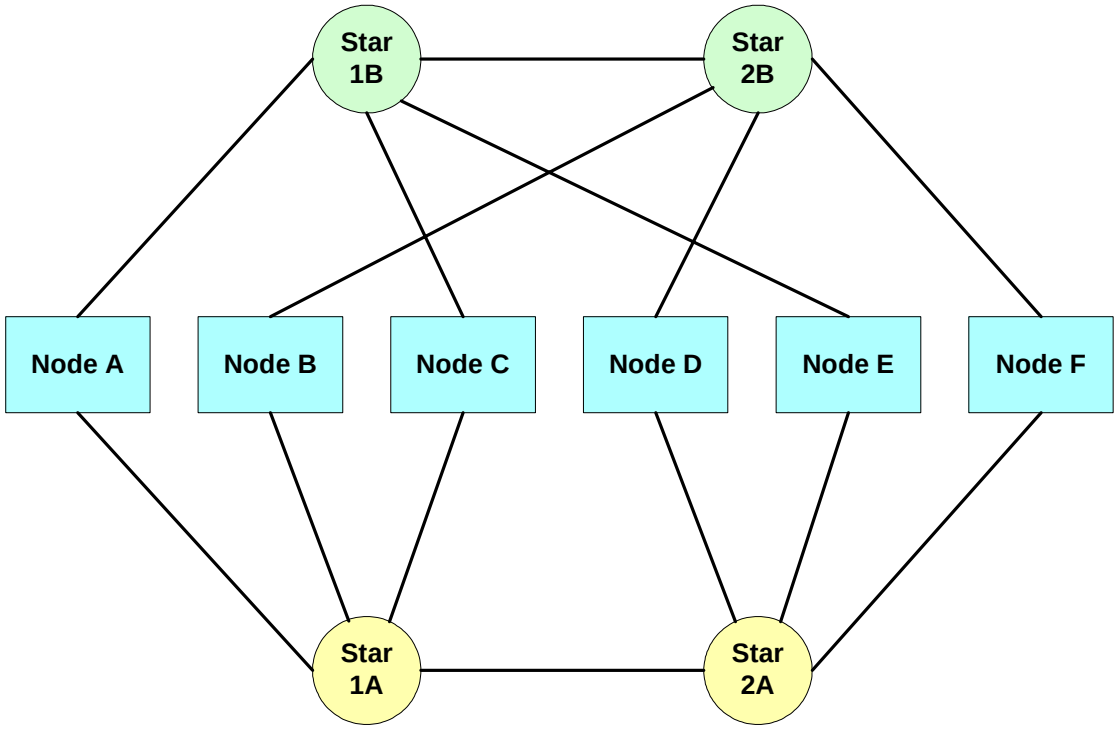

Figure 1-4: Dual channel cascaded star configuration.

1.8.3 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 star couplers also limits the number of cascaded star couplers in a hybrid topology.

There are a large number of possible hybrid topologies, but only two representative topologies are shown here. Figure 1-5 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 a star coupler. Other nodes (nodes E, F, and G) are connected to each other using a bus topology. This bus is also connected to a star coupler, allowing nodes E, F, and G to communicate with the other nodes.

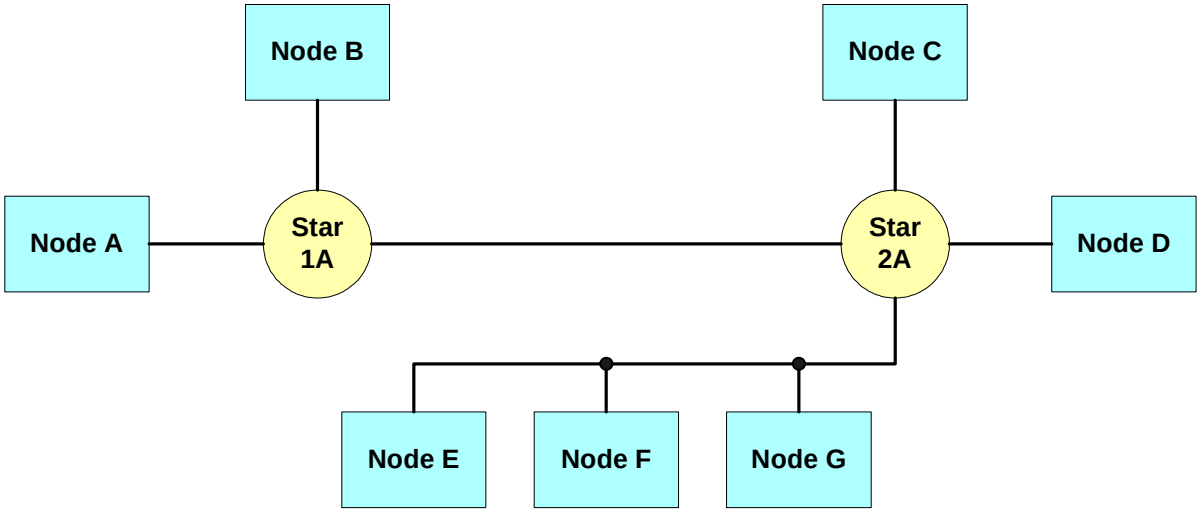

Figure 1-5: Single channel hybrid example.

A fundamentally different type of hybrid topology is shown in Figure 1-6. 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.

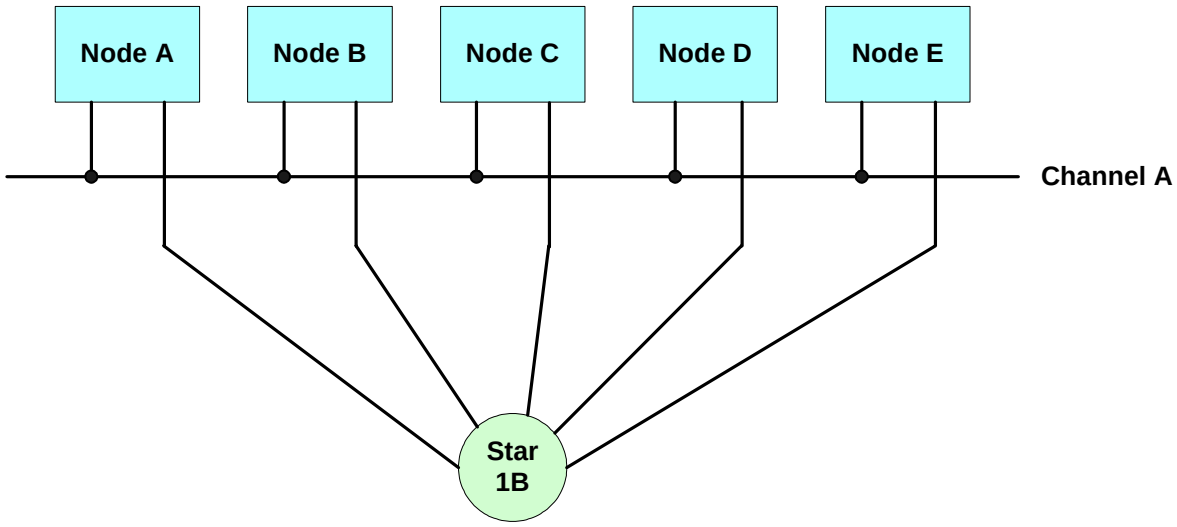

Figure 1-6: Dual channel hybrid example.

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 currently recommended.

1.9 Example node architecture

1.9.1 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 [EPL05]; references are made here to appropriate text passages from this document.

1.9.2 Overview

Figure 1-7 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.

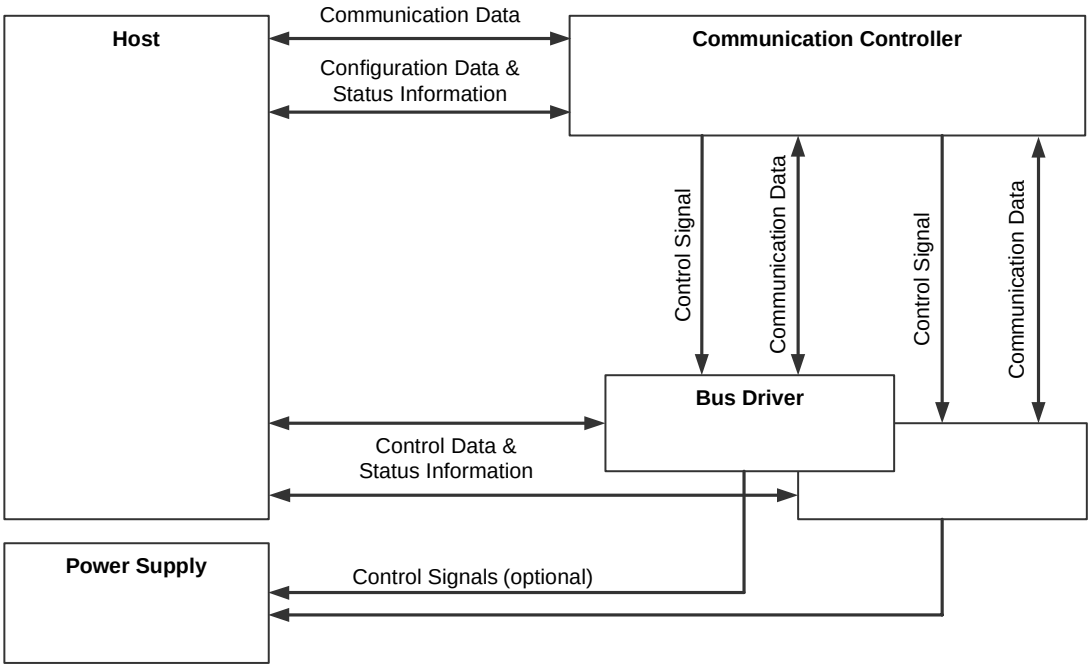

Figure 1-7: Logical interfaces.

1.9.3 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.

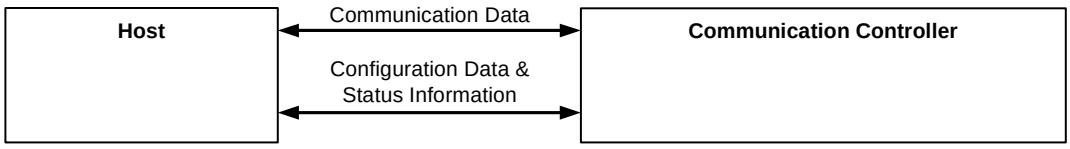

Figure 1-8: Host - communication controller interfaces.

1.9.4 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).

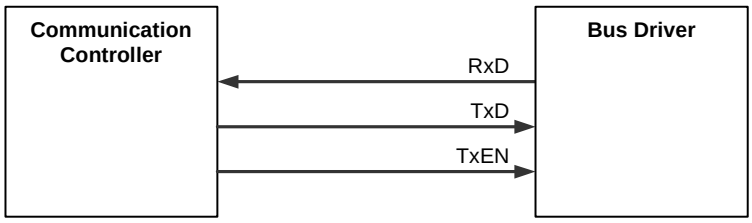

Figure 1-9: Communication controller - bus driver interface.

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.

The electrical characteristics and timing of these signals are specified in [EPL05].

1.9.5 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 1-10) or by a Serial Peripheral Interface (SPI) (see option B in Figure 1-11).

1.9.5.1 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 supports control of optional operational modes.

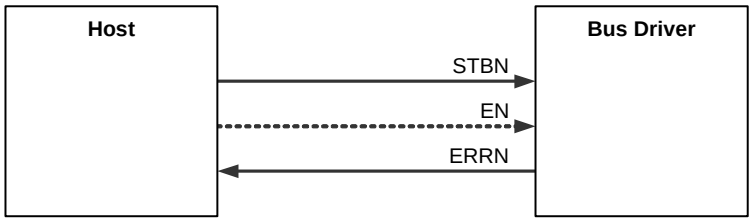

Figure 1-10: Example bus driver - host interface (option A).

This interface is product specific; some restrictions are given in [EPL05] that define minimum functionality to ensure interoperability.

1.9.5.2 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 [EPL05].

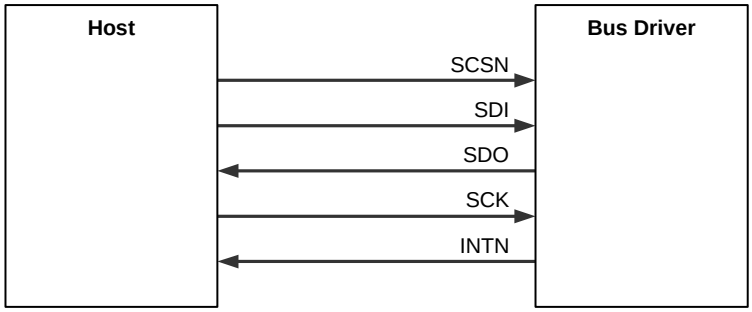

Figure 1-11: Example bus driver - host interface (option B).

1.9.6 Bus driver - power supply interface (optional)

The inhibit signal (INH) 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 1-12: Bus driver - power supply interface.

The electrical characteristics and behavior of the INH signal are specified in [EPL05].

1.10 Testability Requirements

The FlexRay Conformance Test Specification [DLLCT05] 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.

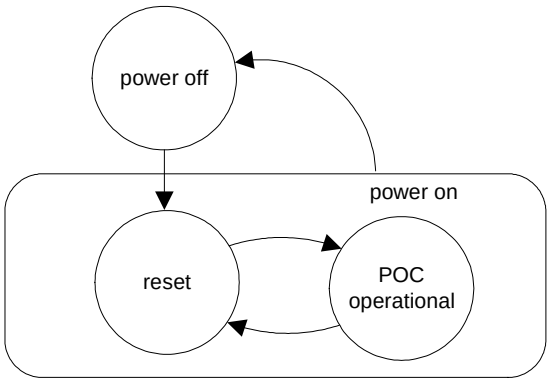

Figure 2-1: Power moding of the communication controller.

Figure 2-1 depicts an overview of the CC power moding operation. Power level and time thresholds and the state of the hardware reset dictate its operation. These product-specific thresholds are briefly described in Table 2-1.

Name	Description
PowerOnPowerThreshold	Power level that causes a transition from the *power off* state to the *power on* state
PowerOffPowerThreshold	Power level that causes a transition from the *power on* state to the *power off* state
POCOperationalPowerThreshold	Power level that must be sustained before the *POC operational* state can be entered from the *reset* state
POCOperationalTimeThreshold	The time duration that the power level *POCOperationalPower-Threshold* must be sustained before *POC operational* state can be entered
ResetPowerThreshold	Power level that causes a transition from the *POC operational* state to the *reset* state if sustained
ResetTimeThreshold	The time duration that the power level *ResetPowerThreshold* must be sustained before the transition from the *POC operational* state to the *reset* state is caused

Table 2-1: Communication controller power moding thresholds.

The CC shall transition from *power off* to *power on* when the supply voltage exceeds *PowerOnPower-Threshold* and shall transition from *power on* to *power off* when the power supply decreases below *Power-OffPowerThreshold*.

Upon entry to *power on*, the CC shall initially reside in the *reset* state. The CC shall transition from *reset* to *POC operational* when both of the following two conditions are met:

1. A power level of at least *POCOperationalPowerThreshold* is sustained for a time duration of at least *POCOperationalTimeThreshold*, and,
2. The hardware reset is not asserted.

The CC shall transition from *POC operational* to *reset* when either of the following two conditions are met:

1. A power level of no more than *ResetPowerThreshold* but still greater than *PowerOffPowerThreshold*, is sustained for a time duration of at least *ResetTimeThreshold*, or,
2. The hardware reset is asserted.

In the *power off* state there is insufficient power for the CC to operate[8]. 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[9].

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

[9] 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.

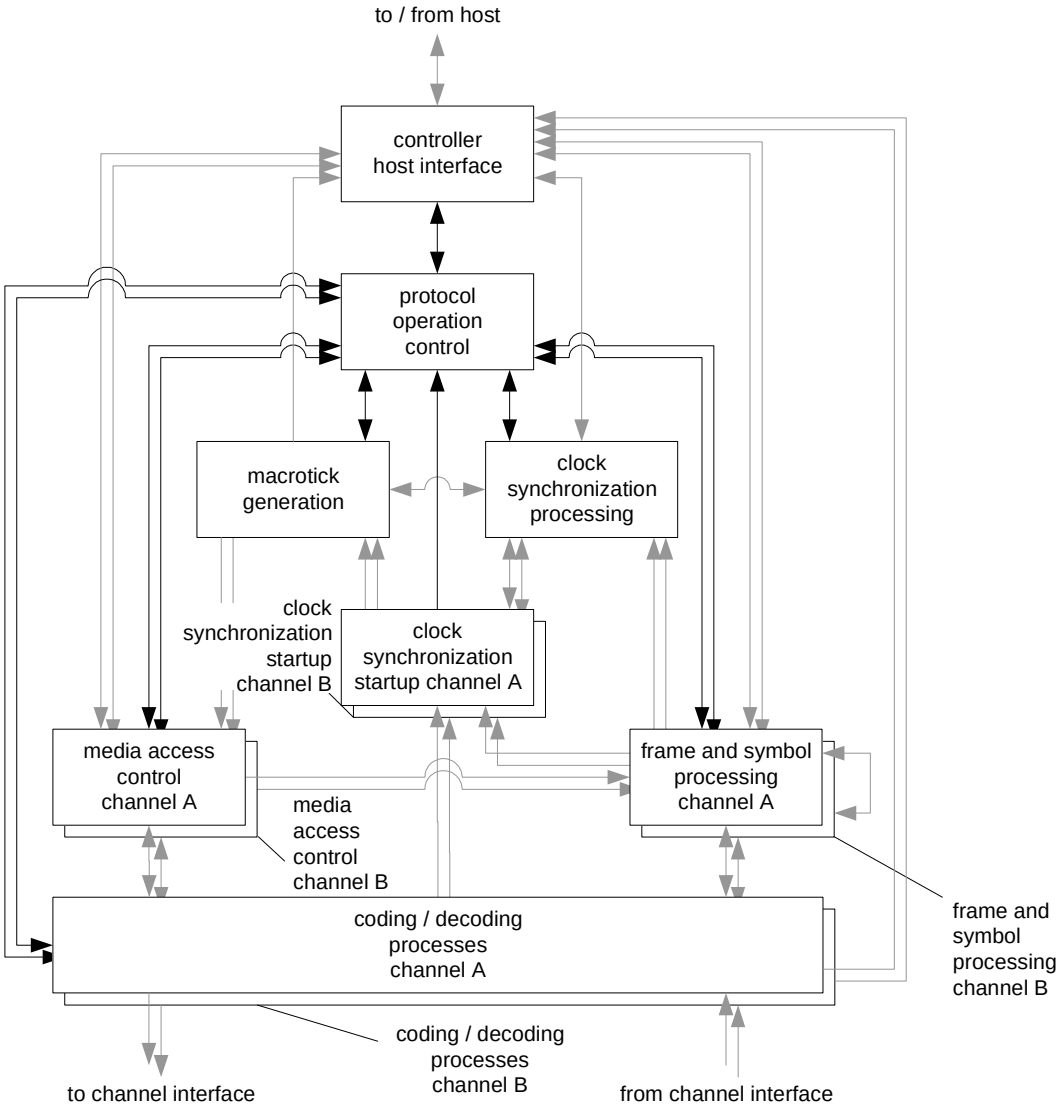

Figure 2-2: Protocol operation control context.

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 2.3.5) 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 2-2.

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*[a]	Immediate
CONFIG	*POC:default config*, *POC:ready*	Immediate
CONFIG_COMPLETE	*POC:config*	Immediate
DEFAULT_CONFIG	*POC:halt*	Immediate
FREEZE	All	Immediate
HALT	*POC:normal active*, *POC:normal passive*	End of cycle
READY	All except *POC:default config*, *POC:config*, *POC:ready*, *POC:halt*	Immediate
RUN	*POC:ready*	Immediate
WAKEUP	*POC:ready*	Immediate

Table 2-2: CHI host command summary.

[a] The ALLOW_COLDSTART command is processed as described in Figure 2-9 except when the POC is in the *POC:integration listen* state, in which case it is processed by the SDL in Figure 7-18.

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 POC or one of the core mechanisms.

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;
      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 eight 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 2.3.6.1). *vPOC!CHIHaltRequest* is Boolean.

vPOC!ColdstartNoise is used to indicate that the STARTUP mechanism completed under noisy channel conditions (see section 7.2). *vPOC!ColdstartNoise* is Boolean.

vPOC!SlotMode is used to indicate what slot mode the POC is in (see sections 2.3.6.1, 2.3.6.2.2, and 2.3.6.2.3). *vPOC!SlotMode* is based on the *T_SlotMode* formal definition in Definition 2-3.

```
newtype T_SlotMode
      literals SINGLE, 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 2.3.6.2.2 and 2.3.6.2.3). *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 2-7 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;
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 2.3.6.2.4.

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 1.8). The POC behavior depicted in this chapter is that of a node connected to both channels. The behavior of a single channel variant is intuitively obvious. Specifically, the POC for this variant only

- Instantiates processes for the attached channel,
- Generates termination signals for the attached channel, and,
- Generates moding signals for the attached channel.

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 for a node attached to only one channel would only instantiate the process set corresponding to the attached channel.

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. The POC for a node attached to only one channel would only generate the signal set corresponding to the attached channel.

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. The POC for a node attached to only one channel would only generate the signal set corresponding to the attached channel.

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[10].

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

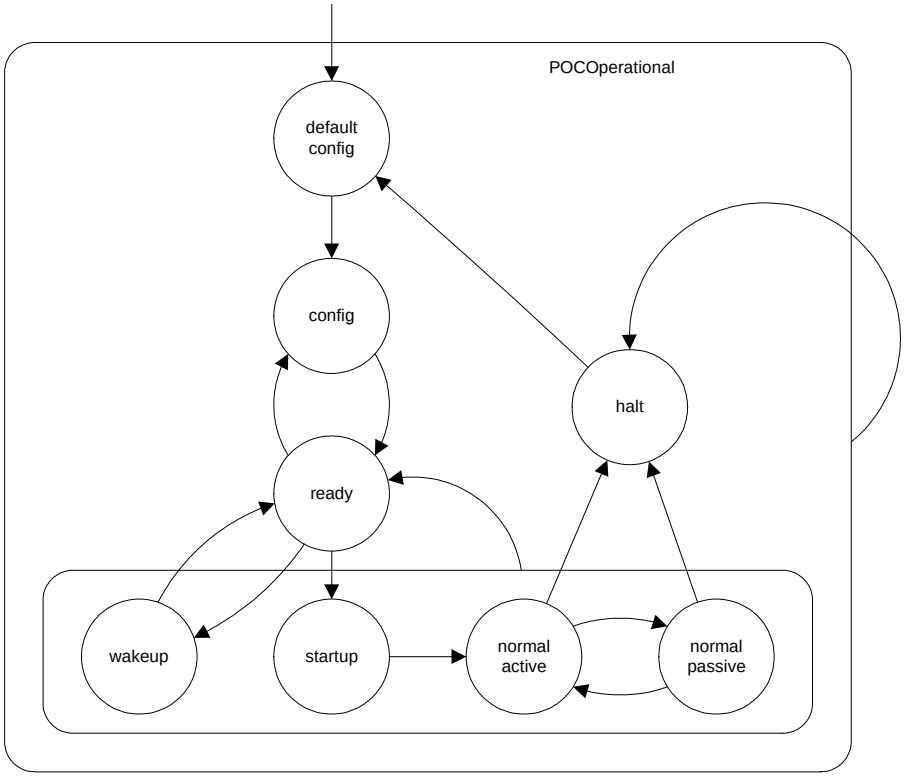

Figure 2-3: Overview of protocol operation control.

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

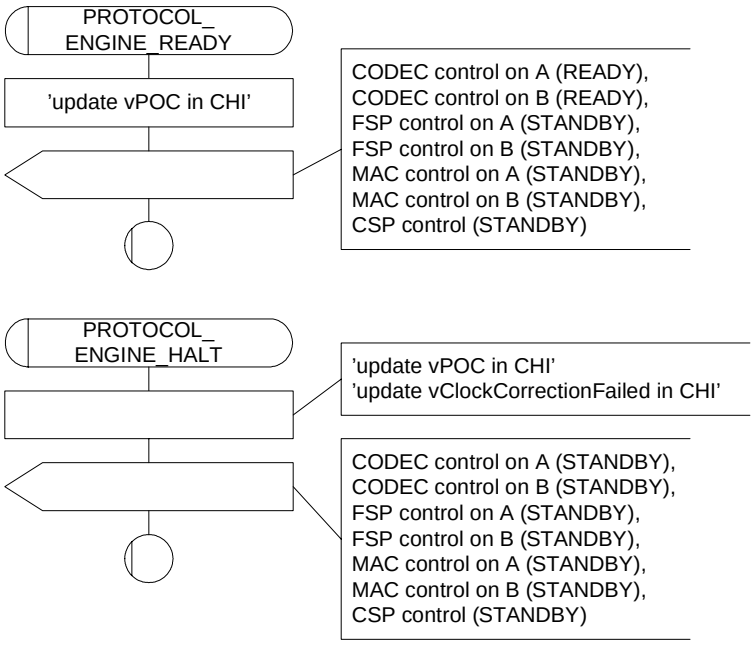

Figure 2-4: Macros to mode the core mechanisms for transitions to the *POC:ready* and *POC:halt* states [POC].

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.

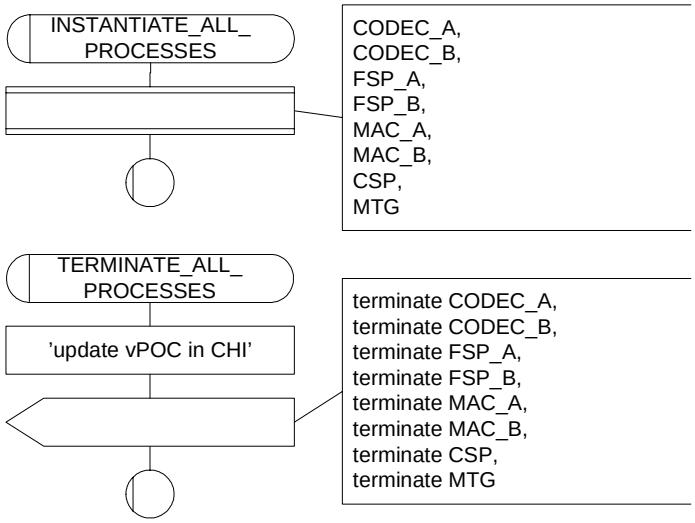

Figure 2-5: Macros for creating and terminating processes [POC].

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 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 2-8) that is detailed in Chapter 7.

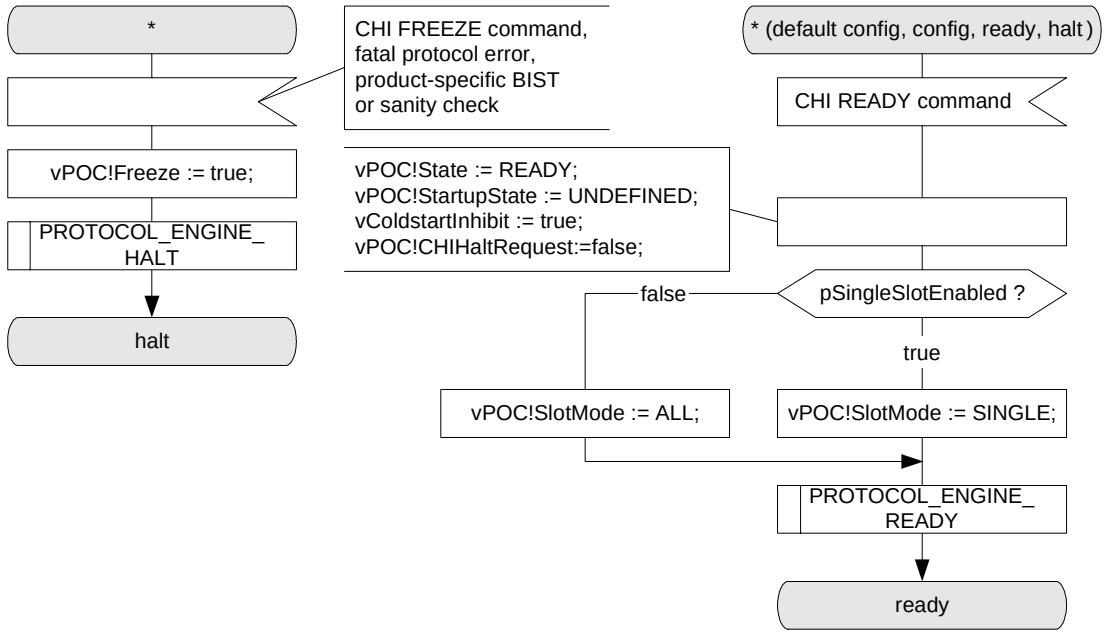

Figure 2-6: POC preempting commands [POC].

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 one of the core mechanisms.
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[11].

At the host's discretion, the ongoing operation of the POC can be interrupted by immediately placing the POC in the *POC:ready* state[12]. In response to this command, the POC modes the core mechanisms appropriately (see section 2.3.1), reinitializes *vColdstartInhibit* and *vPOC!SlotMode*, and then transitions to *POC:ready*.

[11] 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.

2.3.4 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 2-7.

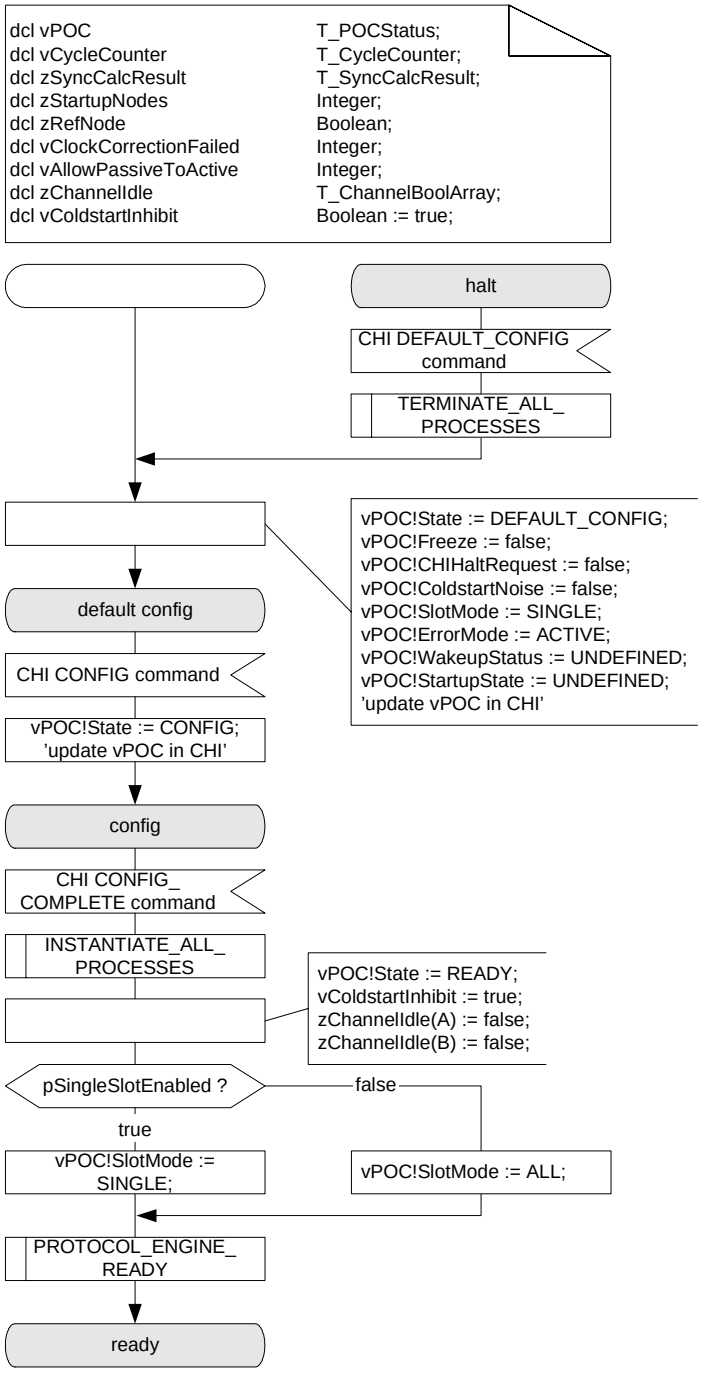

Figure 2-7: Reaching the *POC:ready* state [POC].

[12] The application notes contain several circumstances where the host avails itself of this command to perform control tasks that are jointly performed by the CC and the host. Generally it is a host mechanism for disrupting a CC process that it suspects has failed.

The POC shall enter the *POC:default config* state when the CC enters the *POC Operational* power 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 is 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 2-7. The initial value of *vPOC!SlotMode* is defaulted to SINGLE 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 2.3.5).

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[13], and the initial value of the *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 Reaching the *POC:normal active* state

Following the initialization sequence (see section 2.3.4) the CC resides in the *POC:ready* state (see Figure 2-8). 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.

[13] The value is determined by the node configuration, *pSingleSlotEnabled*, a Boolean used to indicate whether the single slot mode is enabled. This supports an optional strategy to limit frame transmissions following startup to a single designated frame until the host confirms that the node is synchronized to the cluster and enables the remaining transmissions with an ALL_SLOTS command (see section 2.3.6.1).

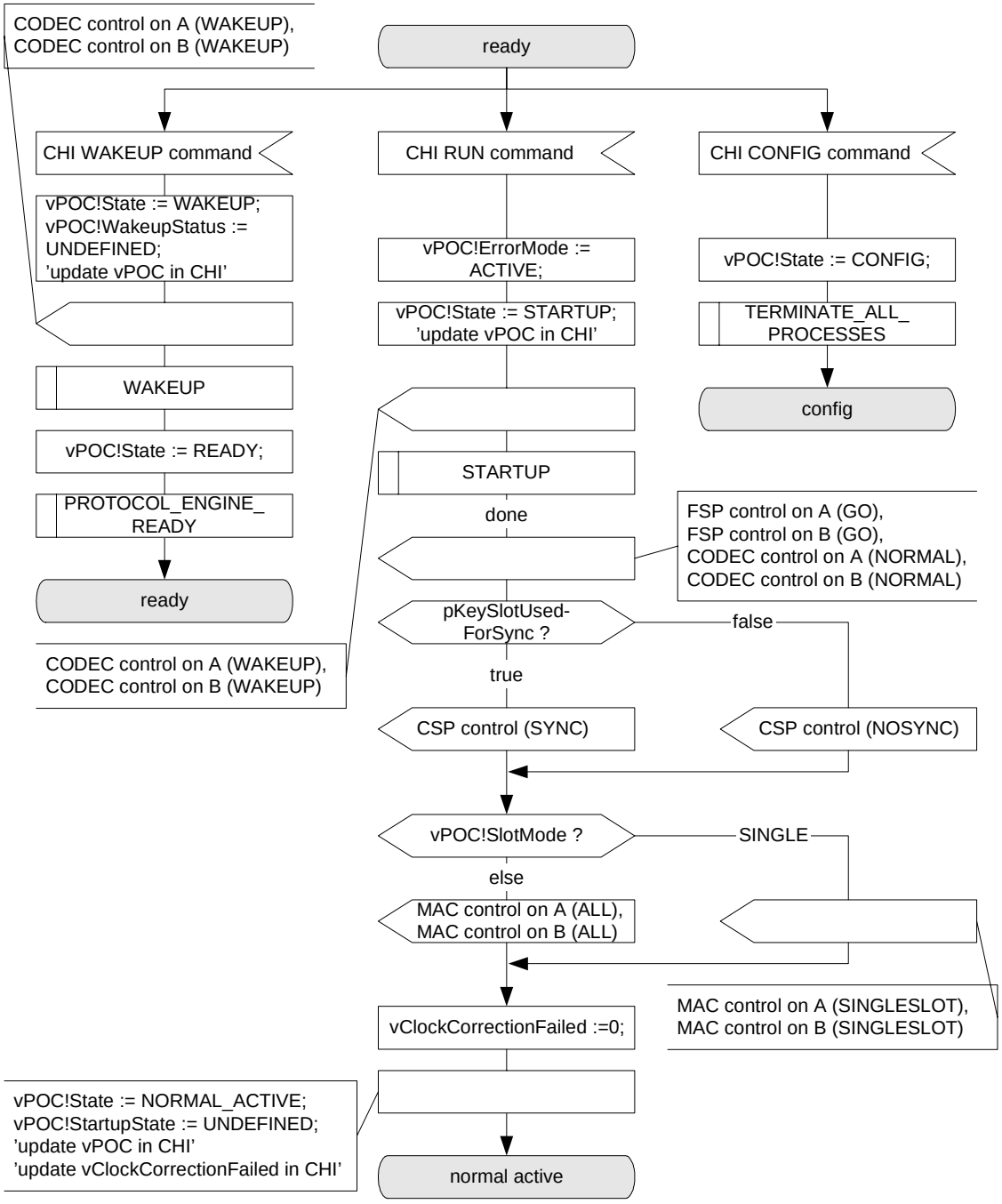

Figure 2-8: POC behavior in preparation for normal operation [POC].

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 2-8 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[14]. Then the startup procedure is executed. In Figure 2-8 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* (see Appendix B) as depicted in Figure 2-8:

1. The CODEC mechanism shall be moded to NORMAL and the FSP mechanism shall be moded to GO for both channels.
2. If the node is a sync node (including coldstart nodes) (*pKeySlotUsedForSync* is true) CSP shall be moded to SYNC mode. Otherwise, CSP shall be moded to NOSYNC.
3. If the node is currently in single slot mode (*vPOC!SlotMode* is SINGLE), then the POC shall mode the MAC to SINGLESLOT mode on both channels. If the node is not currently in single slot 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.

2.3.5.1 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 2-9.

[14] This is necessary because the *POC:ready* state may have been entered due to a 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).

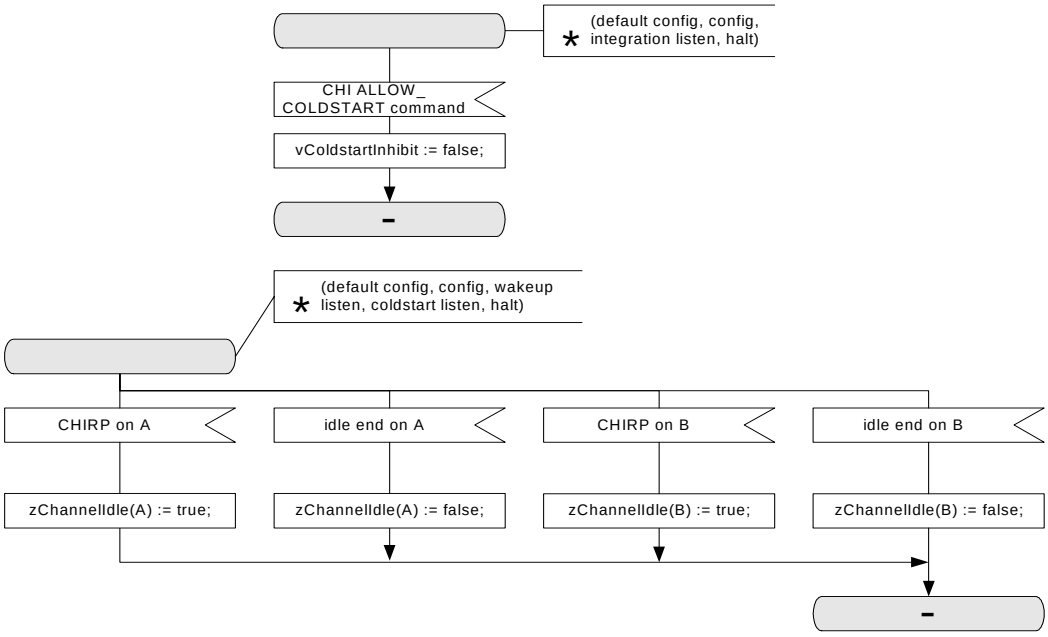

Figure 2-9: Conditions detected in support of the wakeup and startup procedures [POC].[15]

The behavior of the POC during startup is influenced by whether the node is currently inhibited from participating in the startup process as a coldstart node (see section 7.2.3). The node's ability to act as coldstart node is reflected in the Boolean variable *vColdstartInhibit*. While this value is acted upon in the startup procedure, the CHI can change it at any time once the *POC:ready* state is reached. Hence it is relevant in the current context. The POC shall change the value of *vColdstartInhibit* when instructed to do so by the CHI.

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 2-9 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 CODEC (see section 3.2.6.1). 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 CODEC (see section 3.2.6.1).

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

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

[15] *POC:wakeup listen* is a state from the wakeup procedure (see section 7.1) and *POC:coldstart listen* and *POC:integration listen* are states in the startup procedure (see section 7.2).

```
endnewtype;
```

Definition 2-8: Formal definition of T_Channel.

2.3.6 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 asynchronous capture of host commands that the CHI relays to the POC. While the processing of these commands is part of the basic cyclical operation, the asynchronous capture is not.
- The cyclical processing of error status information and host commands at the end of each cycle.

The capture of asynchronous host commands will be discussed first, followed by the cyclical behavior including the processing of the captured asynchronous commands. The preemptive commands are described in section 2.3.3.

2.3.6.1 Asynchronous commands

The CHI may relay the HALT and ALL_SLOTS commands from the host at any time while the POC is in the *POC:normal active* or *POC:normal passive* states. Their effect is realized during the processing at the end of the cycle, but it is necessary to capture indications that the commands have occurred so that appropriate processing will occur at cycle end. Figure 2-10 depicts the procedures that capture these two commands.

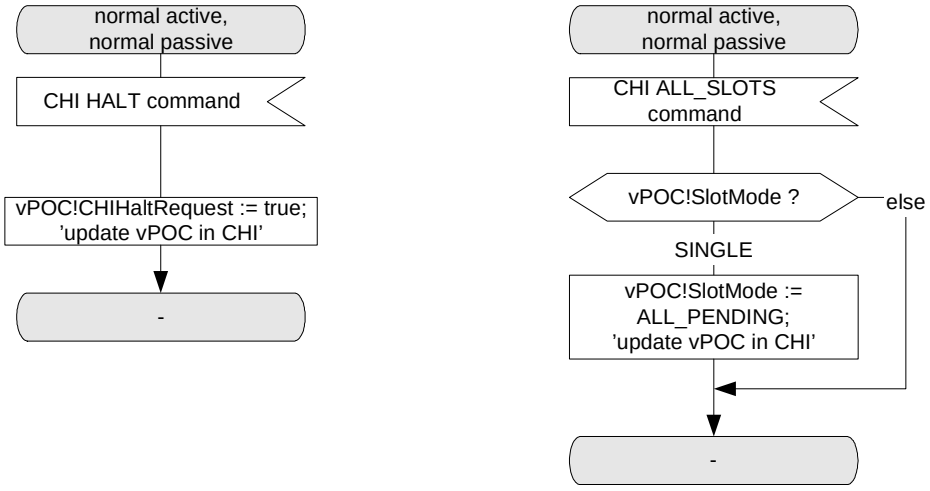

Figure 2-10: Capture of asynchronous host commands for end-of-cycle processing [POC].

The HALT command shall be captured by setting the *vPOC!CHIHaltRequest* value to true. When processed at the end of the current cycle, the 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 ALL_SLOTS command shall be captured by setting *vPOC!SlotMode* to ALL_PENDING. The command shall be ignored if *vPOC!SlotMode* is not SINGLE. 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.

2.3.6.2 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 HALT command.

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.

2.3.6.2.1 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 2-11.

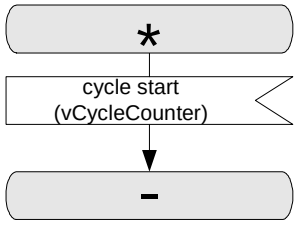

Figure 2-11: POC determination of the cycle counter value [POC].

2.3.6.2.2 *POC:normal active* state

Following a successful startup the POC will reside in the *POC:normal active* state (see section 2.3.5). As depicted in Figure 2-12 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 8-3) signals the end of each communication cycle 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 2.3.6.1).
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 2-12. The details of this task sequence are described in section 2.3.6.2.4.2, but it results in one of three possible POC outcomes depending on the new value of *vPOC!ErrorMode*:
   a. If the *vPOC!ErrorMode* is ACTIVE and the CHI did not relay a HALT command to the POC in the preceding communication cycle (see section 2.3.6.1), the POC shall return to the *POC:normal active* state.
   b. If the *vPOC!ErrorMode* is PASSIVE and the CHI did not relay a HALT command to the POC in the preceding communication cycle (see section 2.3.6.1), 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 or the CHI did relay a HALT command to the POC in the preceding communication cycle (see section 2.3.6.1), the POC shall halt the execution of the core mechanisms by moding them to STANDBY and transition to the *POC:halt* state.

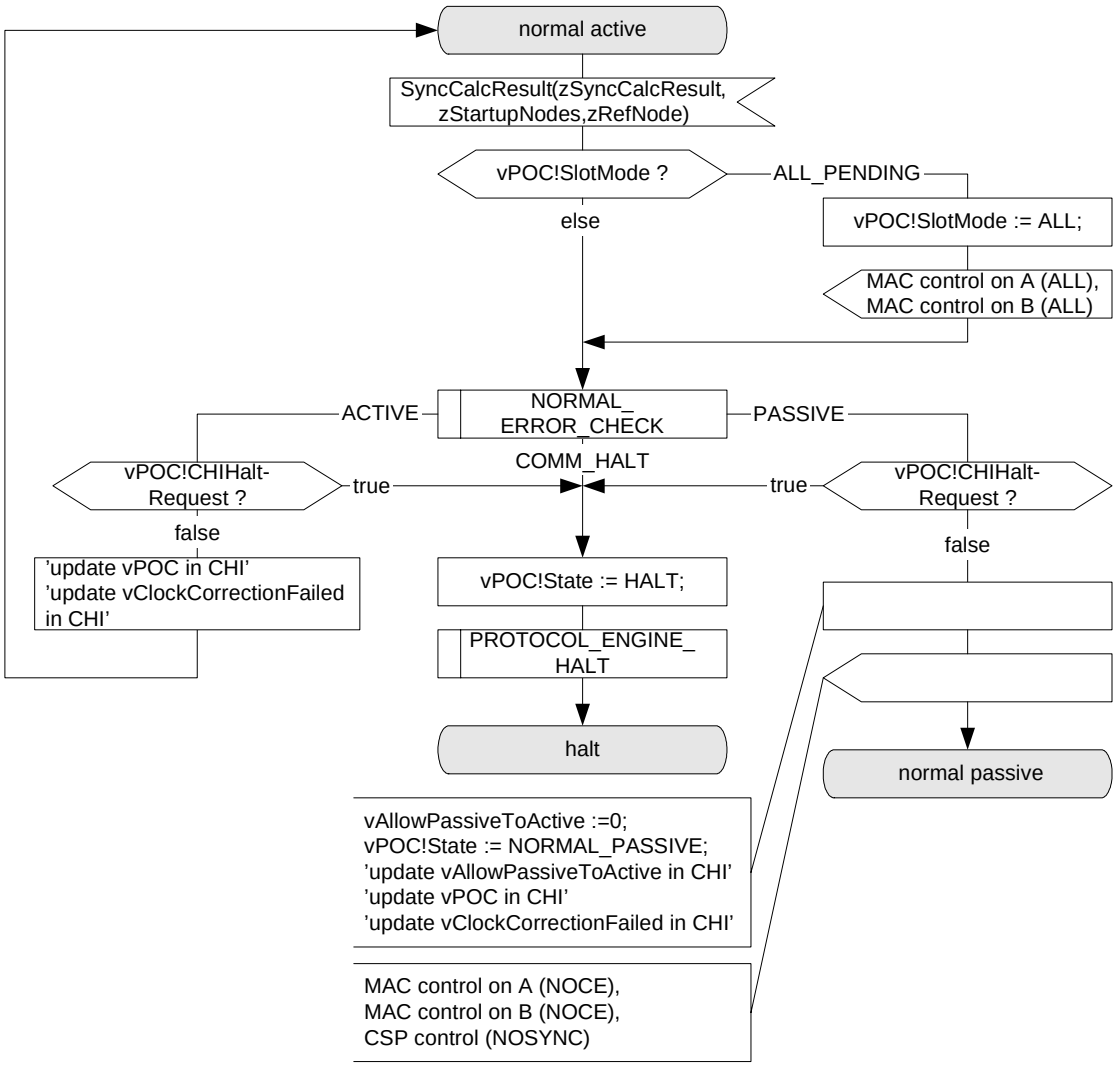

Figure 2-12: Cyclical behavior in the *POC:normal active* state [POC].

2.3.6.2.3 *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 2.3.6.2.2). As depicted in Figure 2-13 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 8-3) signals the end of each communication cycle 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 2.3.6.1).
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 2-13. The details of this task sequence are described in

section 2.3.6.2.4.3, but it results in one of three possible POC outcomes depending on the new value of *vPOC!ErrorMode*:

a. If the *vPOC!ErrorMode* is ACTIVE and the CHI did not relay a HALT command to the POC in the preceding communication cycle (see section 2.3.6.1), 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 single slot mode, and then transition to the *POC:normal active* state.

b. If the *vPOC!ErrorMode* is PASSIVE and the CHI did not relay a HALT command to the POC in the preceding communication cycle (see section 2.3.6.1), the POC shall return to the *POC:normal passive* state.

c. If *vPOC!ErrorMode* is COMM_HALT or the CHI did relay a HALT command to the POC in the preceding communication cycle (see section 2.3.6.1), the POC shall stop the execution of the core mechanisms by moding them to STANDBY and transition to the *POC:halt* state.

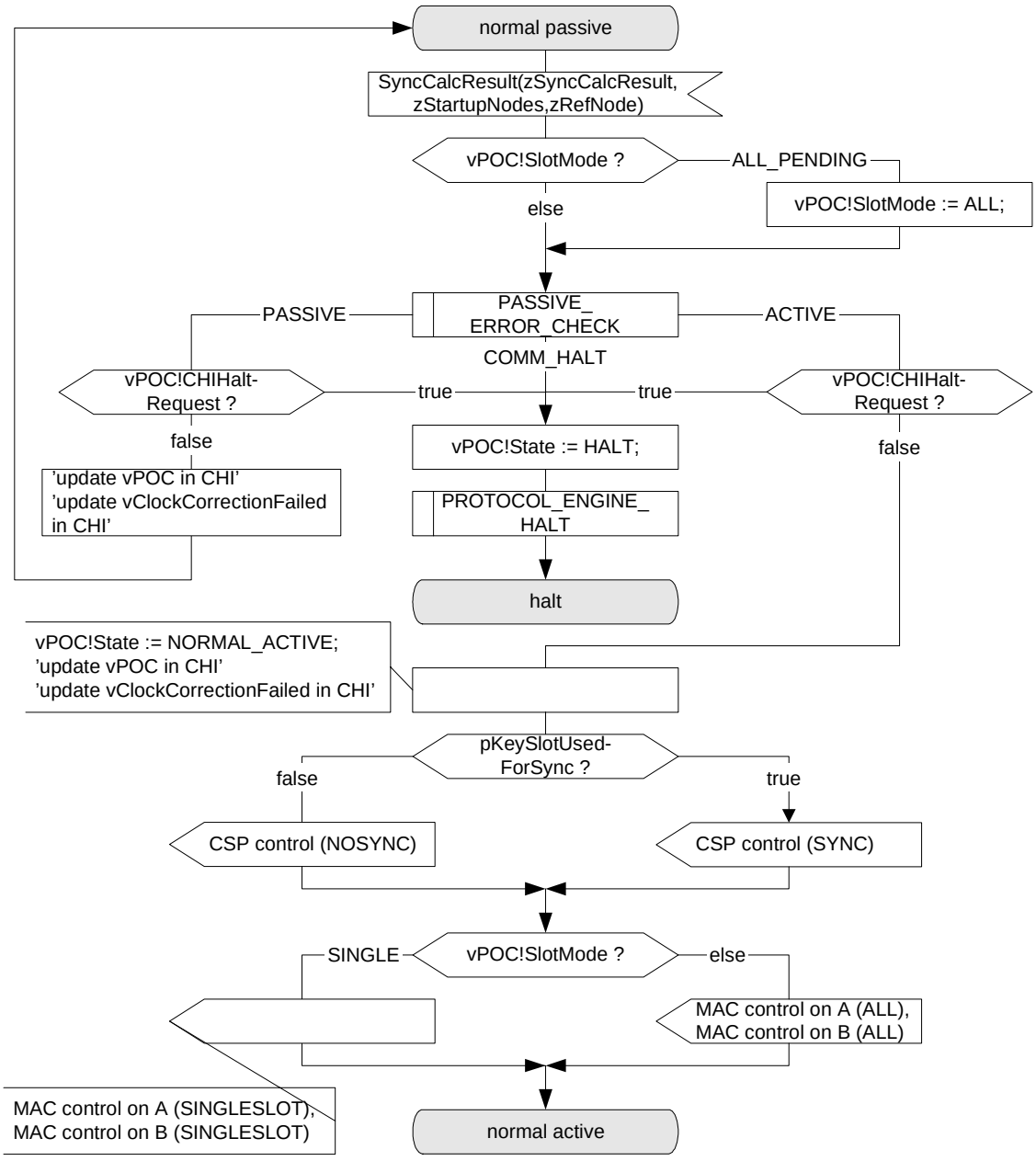

Figure 2-13: Cyclical behavior in the *POC:normal passive* state [POC].

2.3.6.2.4 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 2-14) and PASSIVE_ERROR_CHECK (see Figure 2-15) macros. In both cases, the macro determines a new value of *vPOC!ErrorMode* which, in turn, determines the subsequent cycle-end behavior (see sections 2.3.6.2.2 and 2.3.6.2.3).

2.3.6.2.4.1 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 2.3.6.2.4.2 and 2.3.6.2.4.3. 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.

2.3.6.2.4.2 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 2-14.

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.

The value returned by the macro to the cyclical process depicted in Figure 2-12 corresponds to the new *vPOC!ErrorMode* value.

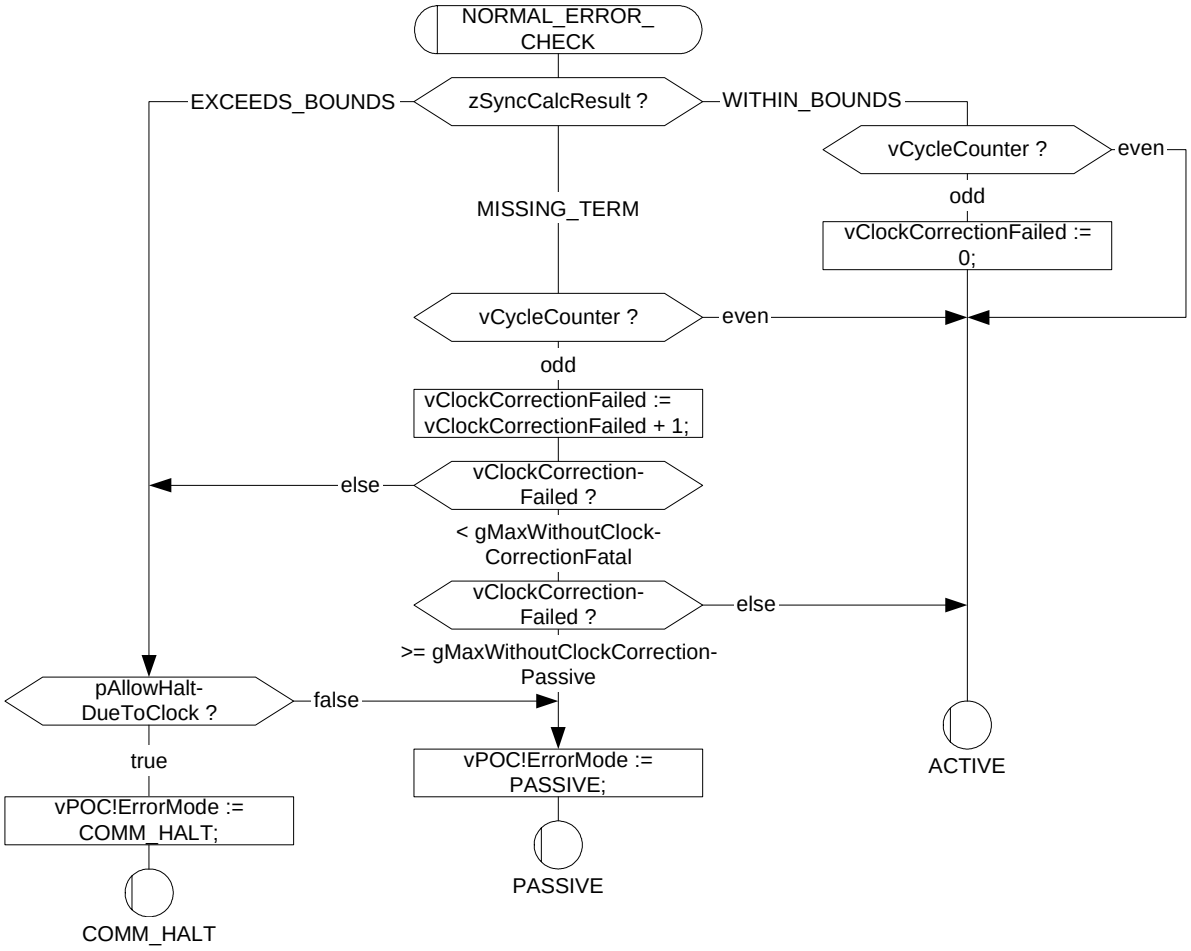

Figure 2-14: Error checking in the *POC:normal active* state [POC].

2.3.6.2.4.3 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 2-15.

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 nonzero) 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 (*zStartupNodes*) of startup frame pairs observed in the preceding double cycle.

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

B. If the node has seen only one startup frame pair (*zStartupNodes* = 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.

The value returned by the macro to the cyclical process depicted in Figure 2-13 corresponds to the new *vPOC!ErrorMode* value.

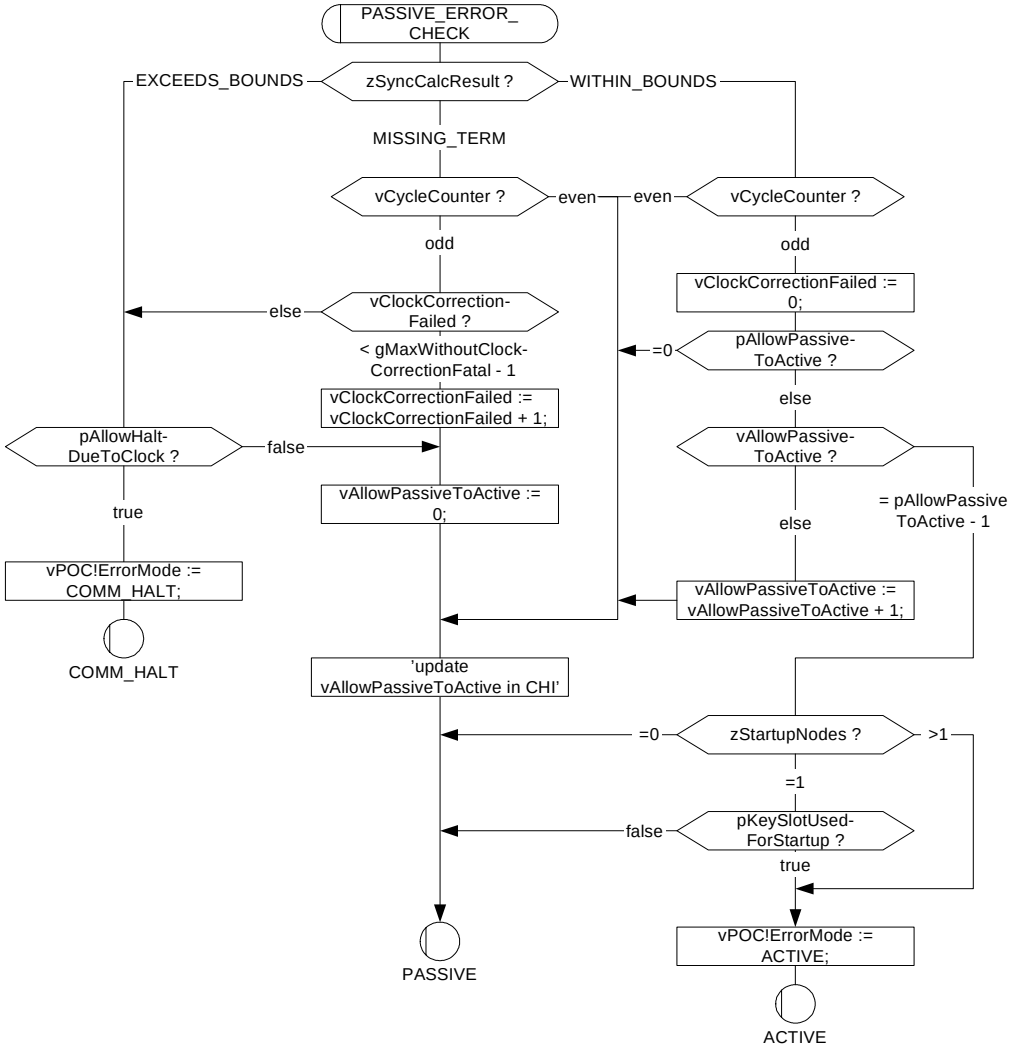

Figure 2-15: Error checking in the *POC:normal passive* state [POC].

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 1.9.4 for additional information on the interface between the CC and the BD. Since the FlexRay protocol is independent from the underlying physical layer, the description in this chapter assumes a binary medium with two distinct levels, called HIGH and LOW.[16] 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.

The relationships between the coding/decoding and the other core mechanisms are depicted in Figure 3-1[17].

[16] Detailed bus state definitions may be found in the appropriate physical layer specification ([EPL05], for example).
[17] 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.

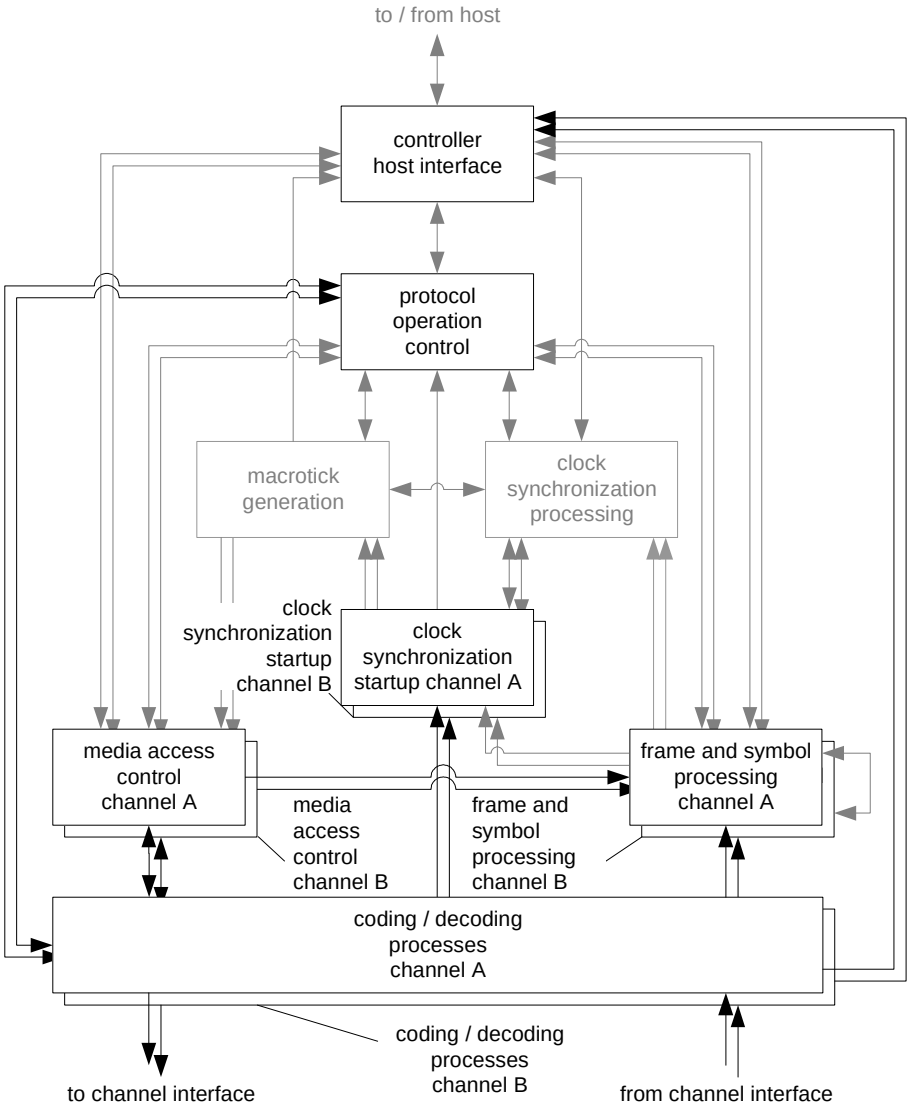

Figure 3-1: Coding / Decoding context

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[18]. A transmitting node generates a TSS that consists of a continuous LOW for a period given by the parameter *gdTSSTransmitter*.

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*[19] and prevents premature channel idle detection[20] 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[21]. 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.

---

[18] The purpose of the TSS is to "open the gates" of an active star, i.e., to cause the star to properly set up input and output connections. During this set up, an active star truncates a number of bits at the beginning of a communication element. The TSS prevents the content of the frame or symbol from being truncated.

[19] See also Chapter 5.

[20] See also section 3.2.4.

[21] 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 3-18) 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 3-18).

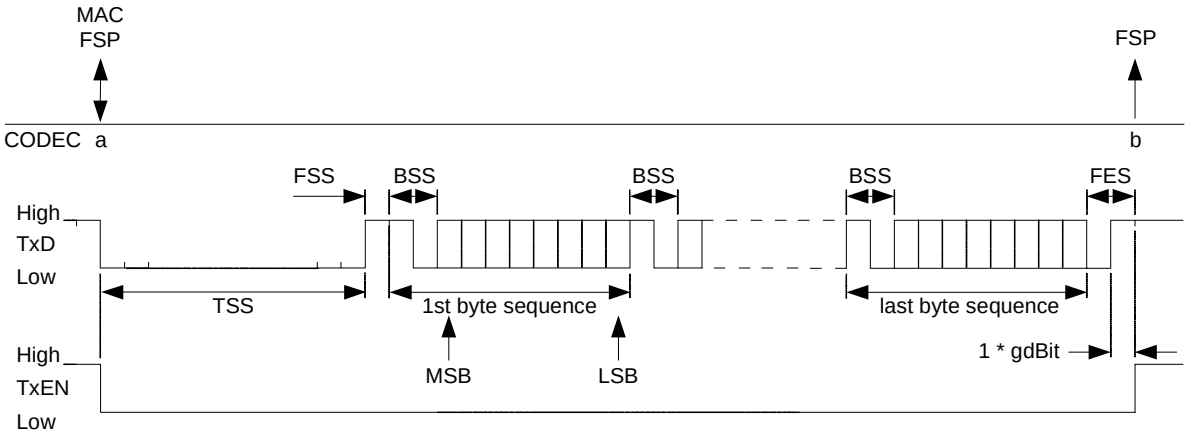

Figure 3-2: Frame encoding in the static segment.

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 (vType, vTF)* received from the MAC process (see Figure 5-17) and output signal *decoding halted on A* sent to the FSP process (see Figure 6-9, Figure 6-10, and Figure 6-17).

b. Output signal *decoding started on A* sent to the FSP process (see Figure 6-18).

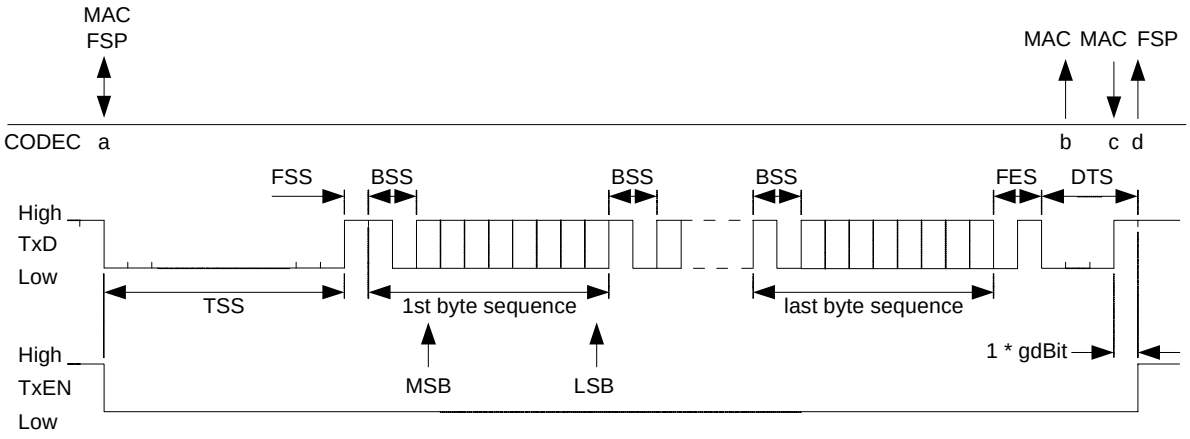

Figure 3-3: Frame encoding in the dynamic segment.

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 (vType, vTF)* received from the MAC process (see Figure 5-21) and output signal *decoding halted on A* sent to the FSP process (see Figure 6-9, Figure 6-10, and Figure 6-17).

b. Output signal *DTS start on A* sent to the MAC process (see Figure 5-21).

c. Input signal *stop transmission on A* received from the MAC process (see Figure 5-21).

d. Output signal *decoding started on A* sent to the FSP process (see Figure 6-18).

3.2.1.2 Symbol encoding

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

1. Pattern 1 = Collision Avoidance Symbol[22] (CAS) and Media Access Test Symbol (MTS).
2. Pattern 2 = Wakeup Symbol (WUS).

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 signal. The function **prepCASstream** used in the CODEC process (see Figure 3-18) prepares the CAS or MTS bit stream for transmission.

[22] See also Chapter 7.

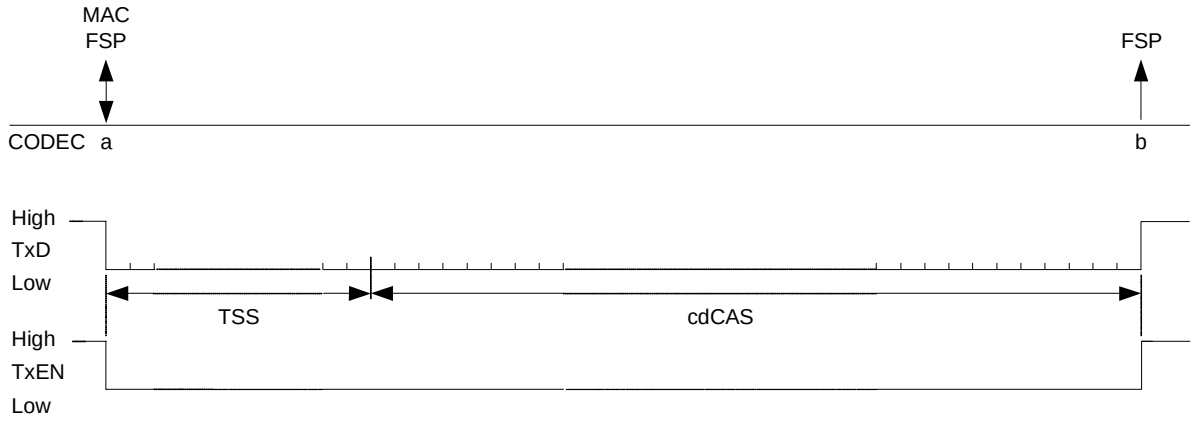

Figure 3-4: CAS and MTS symbol encoding.

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 (vType)* received from the MAC process with *vType* = SYMBOL (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 6-9, Figure 6-10, and Figure 6-17).

b. Output signal *decoding started on A* sent to the FSP process (see Figure 6-18).

3.2.1.2.2 Wakeup symbol

The node shall support a dedicated *wakeup symbol* (WUS) composed of *gdWakeupSymbolTxLow* bits transmitted at a LOW level followed by *gdWakeupSymbolTxIdle* bits of 'idle'. A node generates a *wakeup pattern* (WUP) by repeating the wakeup symbol *pWakeupPattern* times[23]. An example of a wakeup pattern formed by a sequence of two wakeup symbols is shown in Figure 3-5.

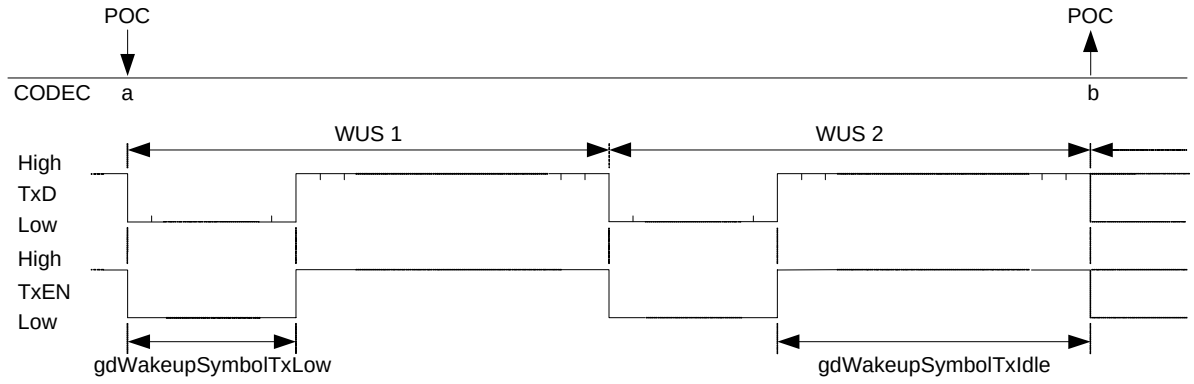

Figure 3-5: Wakeup pattern consisting of two wakeup symbols.

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 (vType)* received from the POC process with *vType* = WUP (see Figure 7-4).

[23] *pWakeupPattern* is a configurable parameter that indicates how many times the WUS is repeated to form a WUP.

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 signal. Note that there is no TSS transmission associated with a WUS. The node must be capable of detecting activity on the channel during *gdWakeupSymbolTxIdle* inside a WUP as shown in Figure 3-6.

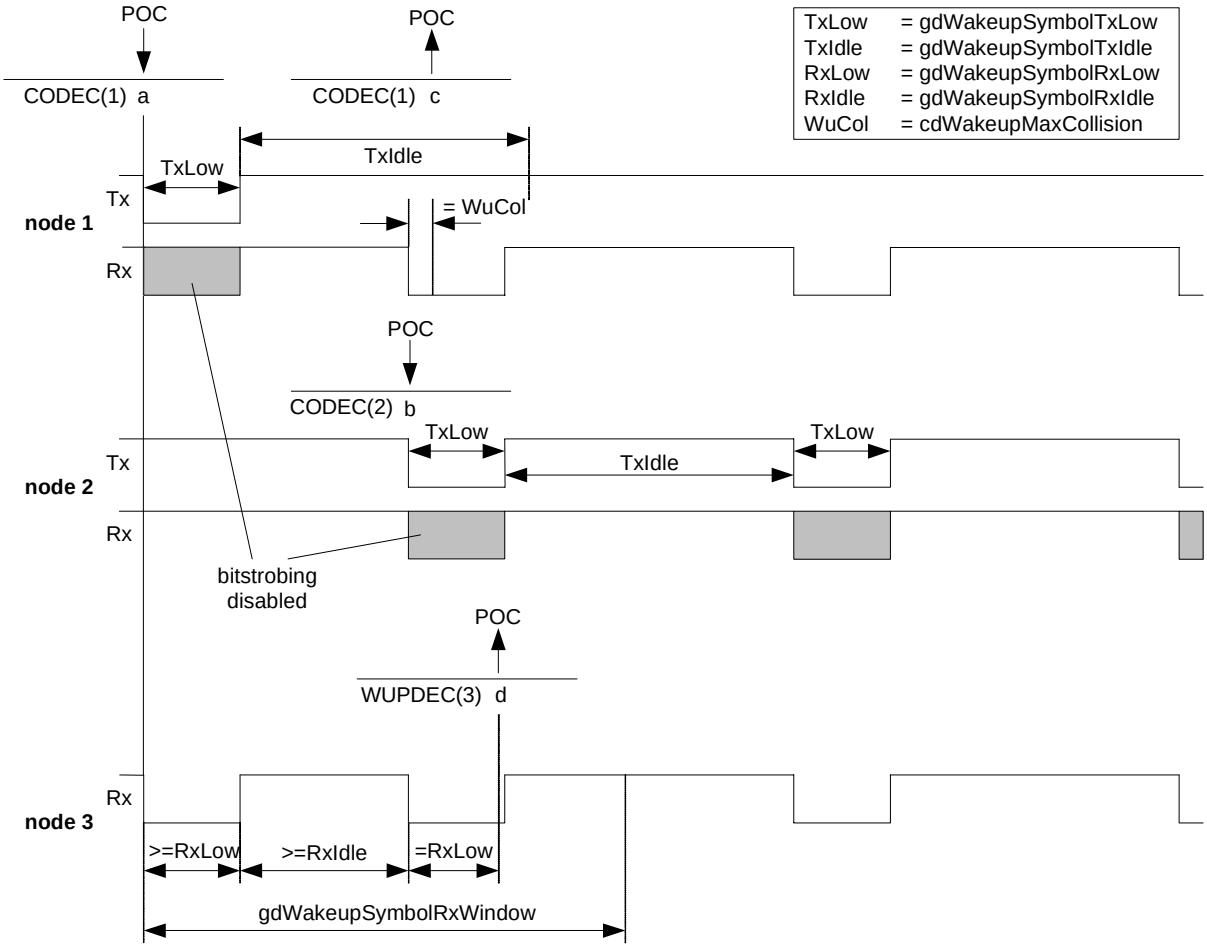

Figure 3-6: Wakeup symbol collision and wakeup pattern reception.

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:[24]

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

b. Input signal *transmit symbol on A (vType)* received from the POC process of node 2 with *vType* = 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 *WUP decoded on A* to the POC process of node 3 (see 3.2.6.2.2).

[24] 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 previously sent by node 1. One possible reason that this could occur is that the first WUS was used to wake up a sleeping star positioned between node 1 and node 2. On the other hand, node 3 may receive the first WUS sent by node 1 if both nodes are on the same branch/stub of the star.

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[25]. The node shall temporarily store the most recent *cVoting-Samples* 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.

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 3-35.

Figure 3-7 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 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* 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 3-35) and shall not directly consider the RxD serial data input signal.

Note: The SDL signal *voted RxD on A* is generated by a sampling event of RxD, and conveys the value *zVotedVal*, which is the result of the majority voting process. In SDL, this process is assumed to take zero time. In an actual implementation, however, there would be some delay between the sample and the computation of the majority voting result. This delay is not shown in the figure. The effect of any voting or sampling delays on the clock synchronization must be internally compensated by the implementation.

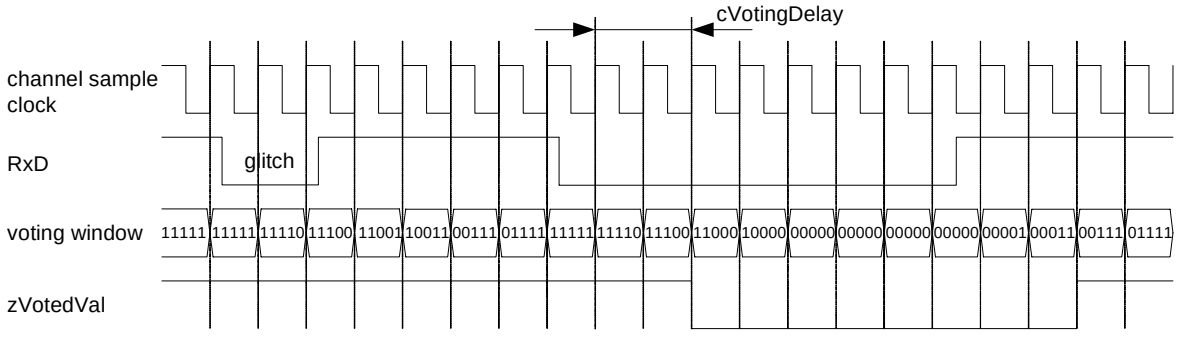

Figure 3-7: Sampling and majority voting of the RxD input.

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.

[25] 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).

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)[26].

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 3-8 shows the mechanism of the bit synchronization when a frame is received.

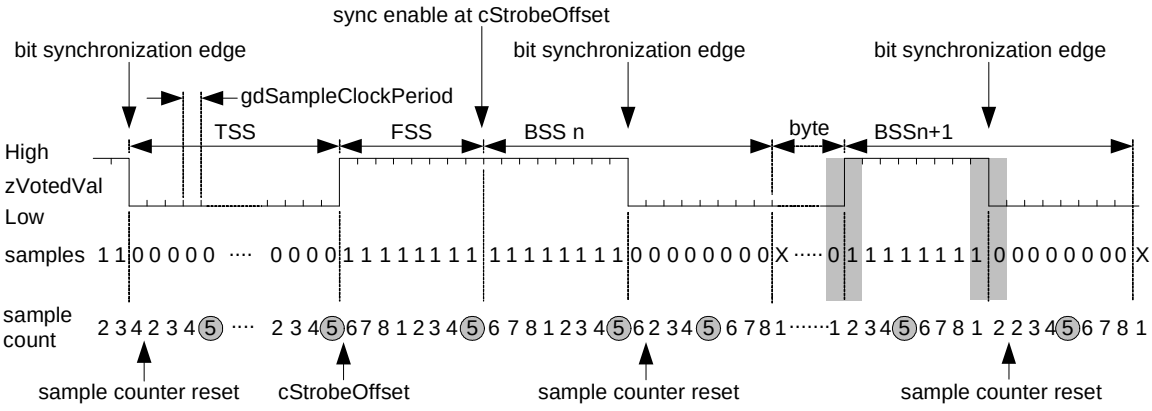

Figure 3-8: Bit synchronization.

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

The example from Figure 3-8 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 3.2.7).

To detect activity on a channel, it is necessary that at least *cStrobeOffset* consecutive LOW samples make it through the majority voter[27]. 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 3-35.

3.2.4 Channel idle detection

The node shall use a channel idle detection mechanism to identify the end of the current communication element. The channel idle detection is context insensitive, i.e., channel idle is detected whenever *cChannelIdleDelimiter* consecutive strobed HIGH bits are decoded on a channel that is not currently considered to be idle. However, 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 created by the POC, it immediately creates 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* consecutive strobed HIGH bits must be decoded to detect that the channel is actually idle (section 3.4.2).

The BITSTRB process is toggled between the GO and STANDBY modes by the CODEC process. When the CODEC process is encoding a communication element, the BITSTRB process is placed in the STANDBY mode and idle detection is terminated. When the encoding of a communication element is concluded, the CODEC places the BITSTRB process in the GO mode where idle detection is reactivated. Whenever the BITSTRB process is placed in the GO mode, the initial assumption of the BITSTRB process is that the channel is active, so *cChannelIdleDelimiter* consecutive strobed HIGH bits must be subsequently decoded to detect that the channel is actually idle.

3.2.5 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[28].

[27] In order to allow the detection of colliding startup frames, a FlexRay system must be configured in such a way that in the event of colliding startup frames a node shall detect at least one strobed LOW bit so that detectable noise is produced to enable setting of the *tWakeupNoise* timer (see Chapter 7).

[28] 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 longer than all other edges of the same frame, causing the TSS seen at the RxD input to be shorter than the TSS that was transmitted. This effect is called TSS truncation and it has various causes (e.g., optical transmission, connection setup in star couplers, etc.). The cumulative effect of all such causes on a TSS transmitted from node M to node N is to reduce the length of the TSS by *dTruncation*$_{M,N}$. A node shall accept the TSS as valid if any number of consecutive strobed logical LOW bits in the range of 1 to (*gdTSSTransmitter* + 1) is detected.

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 3-9 depicts the effect of propagation delay and TSS truncation. For a more detailed description refer to [EPL05].

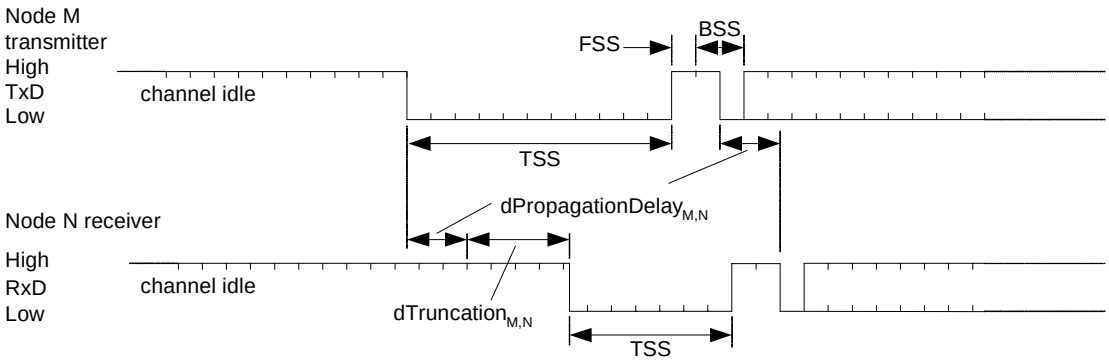

Figure 3-9: TSS truncation and propagation.

As a result of TSS truncation 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 truncation. 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 truncation 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.

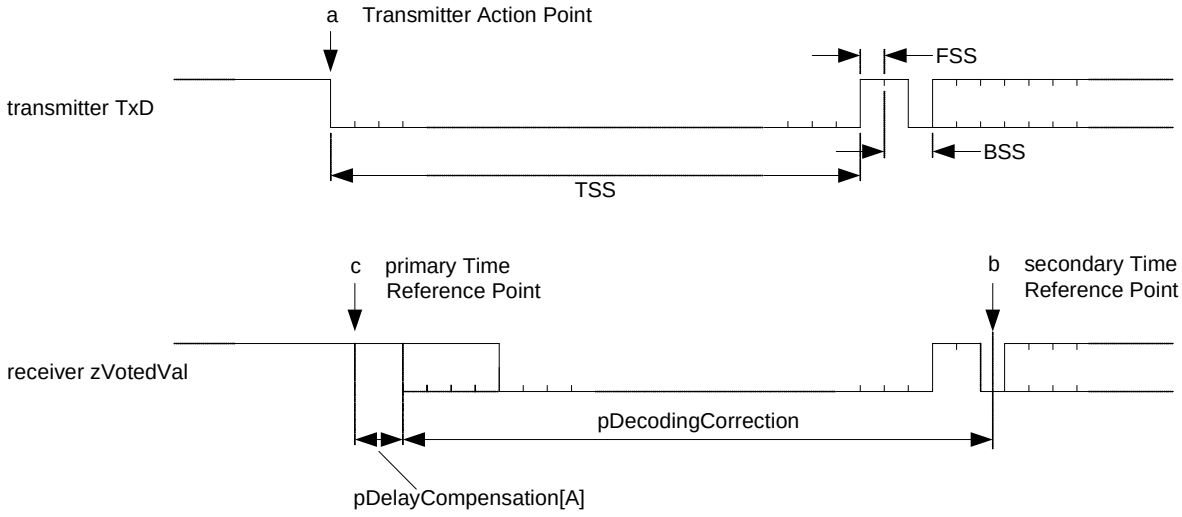

Figure 3-10: Time reference point definitions.

Figure 3-10 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.4.23.

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 3-34, Figure 6-8, Figure 6-10, Figure 6-11, and Figure 8-11). 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.

3.2.6 Frame and symbol decoding

This section specifies the mechanisms used to perform bit stream decoding. The CODEC process interprets the bit stream observed at the voted RxD input of a node. The decoding process controls the flow of the received *voted RxD on A* signal from the BD.

The decoding process of a channel does not perform concurrent decoding of frames and symbols, i.e. for a given channel the node shall support only one decoding process (frame or symbol) at a time. For example, once a new communication element is classified as a start of a frame then symbol 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.

The decoding process shall support successful decoding[29] 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.

3.2.6.1 Frame decoding

A frame starts at *CE start* with the first strobed LOW bit after channel idle. The channel idle delimiter refers to the *cChannelIdleDelimiter* consecutive HIGH bit times prior to the CHIRP.

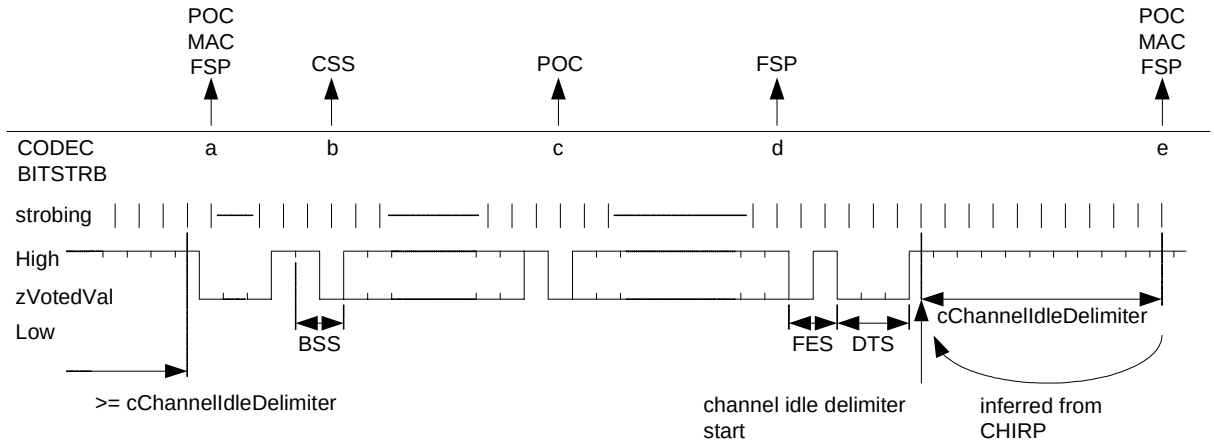

Figure 3-11: Received frame bit stream.

Figure 3-11 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 2-9, Figure 7-3, and Figure 7-11; output signal *CE start on A* to the MAC process shown in Figure 5-21, and to the FSP process shown in Figure 6-9.

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

c. Output signal *header received on A* to the POC process shown in Figure 7-3, Figure 7-5, Figure 7-11, Figure 7-12, and Figure 7-14.

d. Output signal *frame decoded on A (vRF)* to the FSP process shown in Figure 6-10.

e. Output signal *CHIRP on A* to the MAC process shown in Figure 5-21, to the FSP process shown in Figure 6-17, and to the POC process shown in Figure 2-9, Figure 7-3, and Figure 7-11.

[29] 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.

3.2.6.2 Symbol decoding

3.2.6.2.1 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.

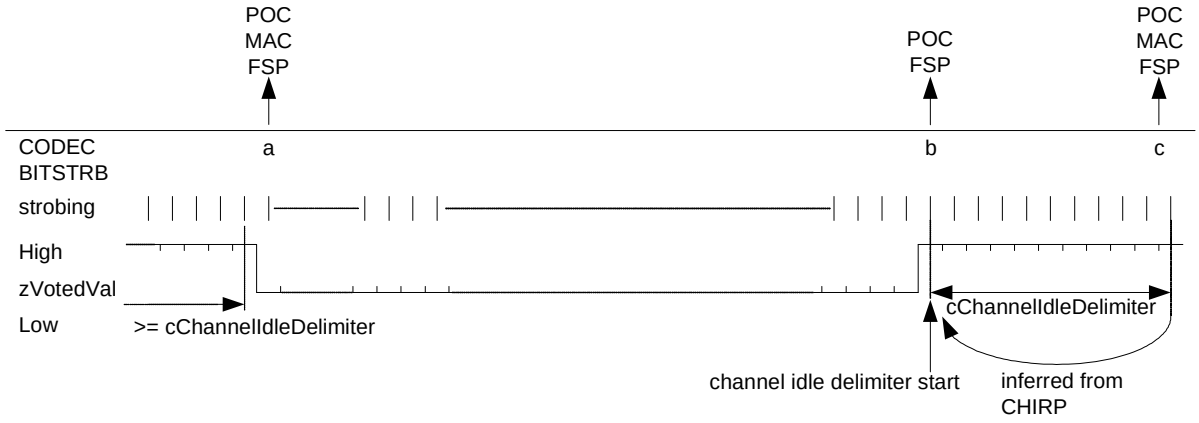

Figure 3-12: Received symbol bit stream.

Figure 3-12 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 2-9, Figure 7-3, and Figure 7-11; output signal *CE start on A* to the MAC process shown in Figure 5-21, and to the FSP process shown in Figure 6-9.

b. Output signal *symbol decoded on A* to the POC process shown in Figure 2-9, Figure 7-3, Figure 7-11, Figure 7-12, and Figure 7-14, to the MAC process shown in Figure 5-21, and to the FSP process shown in Figure 6-11.

c. Output signal *CHIRP on A* to the POC process shown in Figure 2-9, Figure 7-3, and Figure 7-11, to the MAC process shown in Figure 5-21, and to the FSP process shown in Figure 6-17.

3.2.6.2.2 Wakeup symbol decoding

The detection of 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 LOW level with a duration of at least *gdWakeupSymbolRxLow* is detected.
2. This is followed by a duration of at least *gdWakeupSymbolRxIdle* at the HIGH level.
3. This is followed by a duration of at least *gdWakeupSymbolRxLow* at the LOW level.
4. All of the preceding are received within a window with a duration of at most *gdWakeupSymbolRxWindow*.

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

d. Output signal *WUP decoded on A* to the POC process shown in Figure 7-3 and Figure 7-5.

3.2.6.3 Decoding error

Exiting one of the decoding macros SYMBOL_DECODING, FSS_BSS_DECODING, HEADER_DECODING, PAYLOAD_DECODING, or TRAILER_DECODING (shown in Figure 3-24) 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 length 2 bits (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 3-13 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.

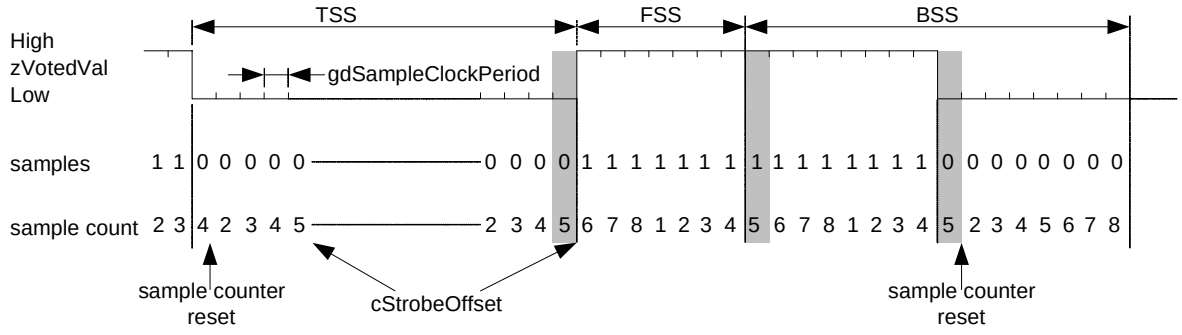

Figure 3-13: Start of frame with FSS BSS decoding.

3.2.7 Signal integrity

In general, there are several conditions (e.g. clock oscillator differences, electrical characteristics of the transmission media or the transceivers, EMC 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.[30]

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

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 [EPL05] in Chapter 12 and in [EPLAN05] in section 2.15.

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, WAKEUP, 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 and the wakeup detection process are executed but the CODEC process waits in the *POC:ready* state.
3. In the NORMAL mode the CODEC process and the bit strobe process are executed, but the wakeup detection process is in standby.
4. In the WAKEUP mode, all processes shall be executed.

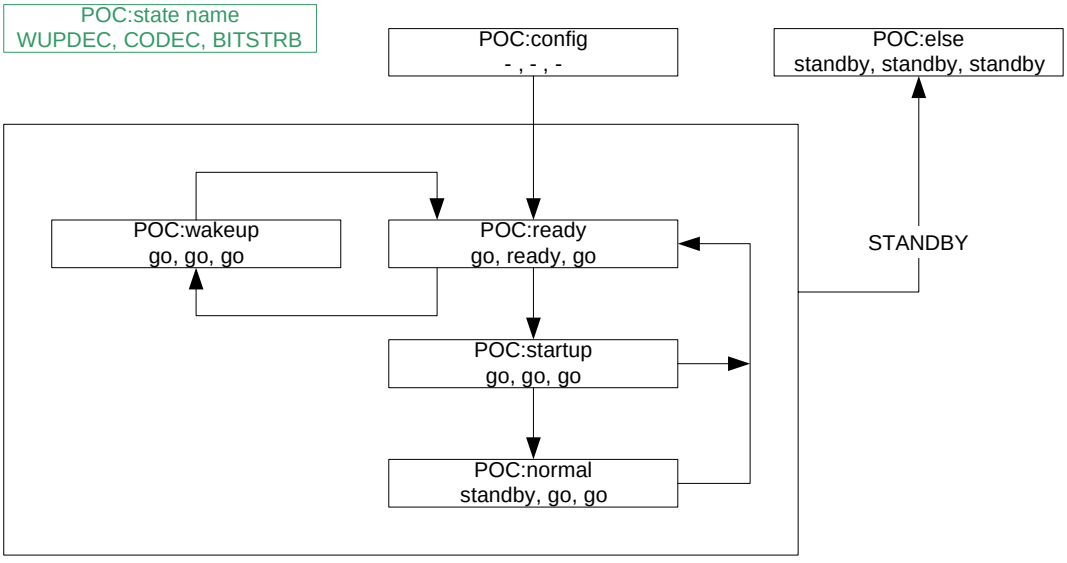

Figure 3-14: Overview of processes and transitions in the encoder/decoder.

3.3.2 Coding and decoding process behavior

This section contains the formalized specification of the CODEC control process. Figure 3-15 and Figure 3-16 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.

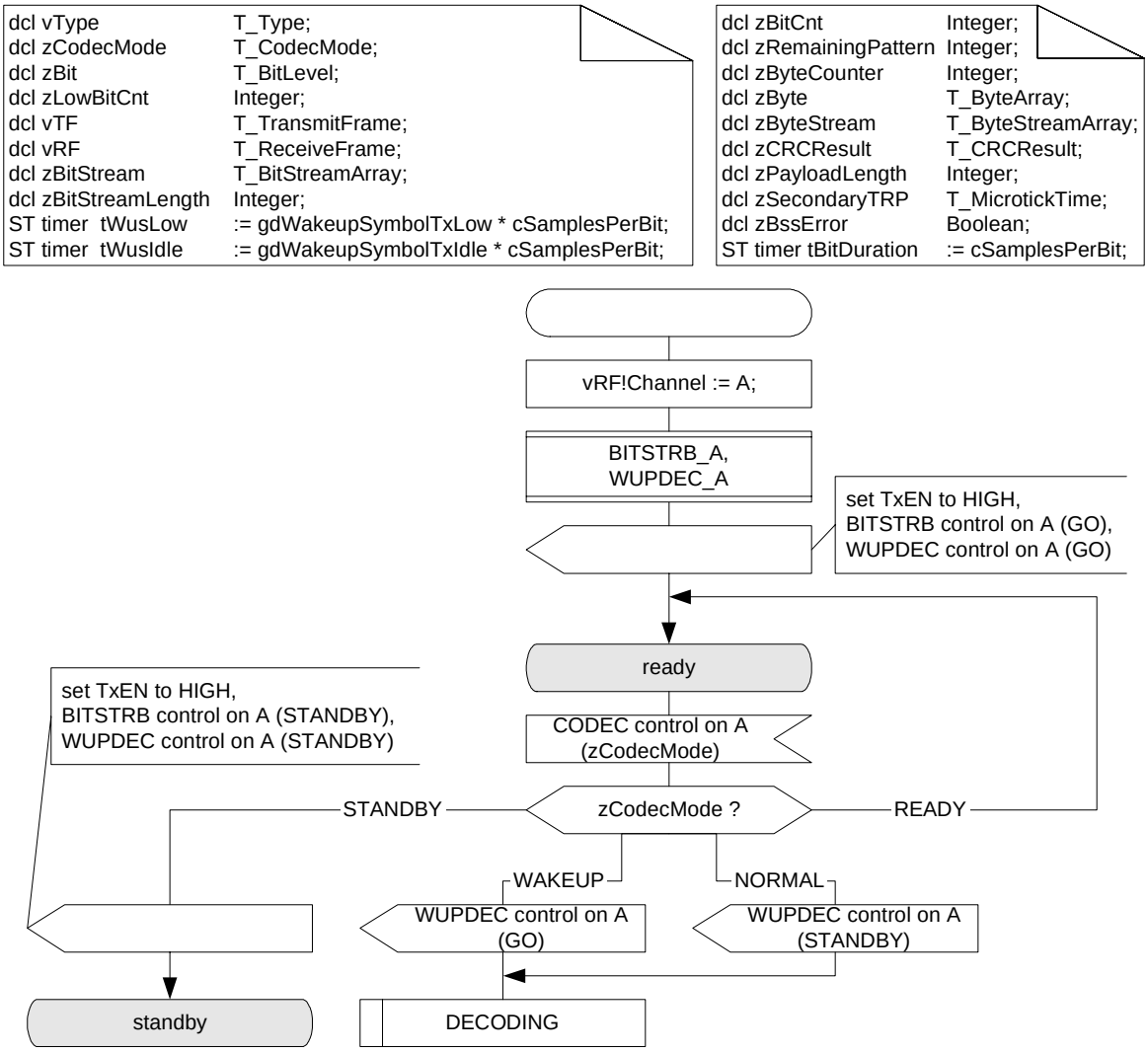

Figure 3-15: CODEC process [CODEC_A].

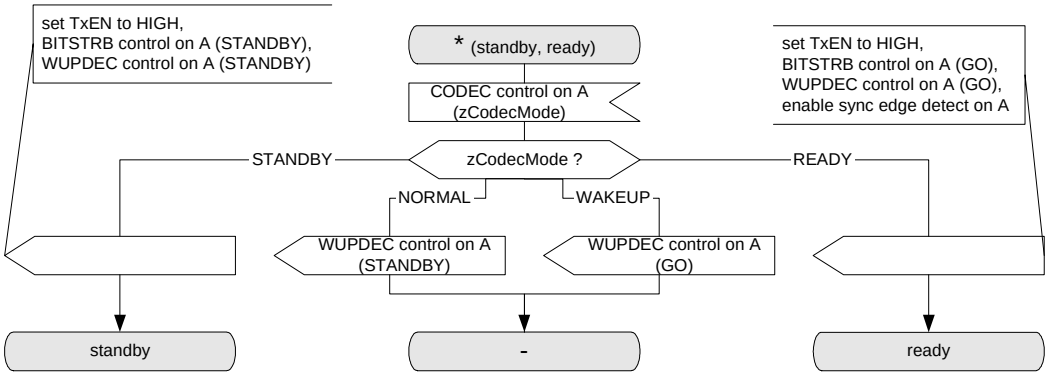

Figure 3-16: Mode control of the CODEC process [CODEC_A].

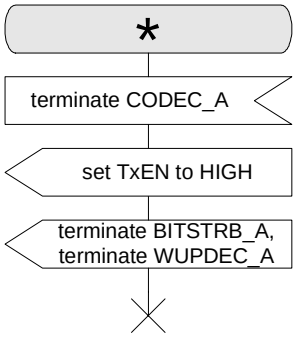

Figure 3-17: Termination of the CODEC process [CODEC_A].

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[31]:

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

Definition 3-2: Formal definition of T_TransmitFrame.[32]

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;
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 to HIGH* and *set TxD to LOW*) and controls the BD via the TxEN signal (the SDL signals *set TxEN to HIGH* and *set TxEN 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 the behavior of the encoding. After reception of the *transmit frame on A* signal the **prepbitstream** function is called. This function 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 assembles a continuous bit stream out of the extended byte sequences. The function **prepCASstream** shall prepend a TSS to the symbol bit stream.

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

[31] 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.

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

```
endnewtype;
```

Definition 3-4: Formal definition of T_BitStreamArray.

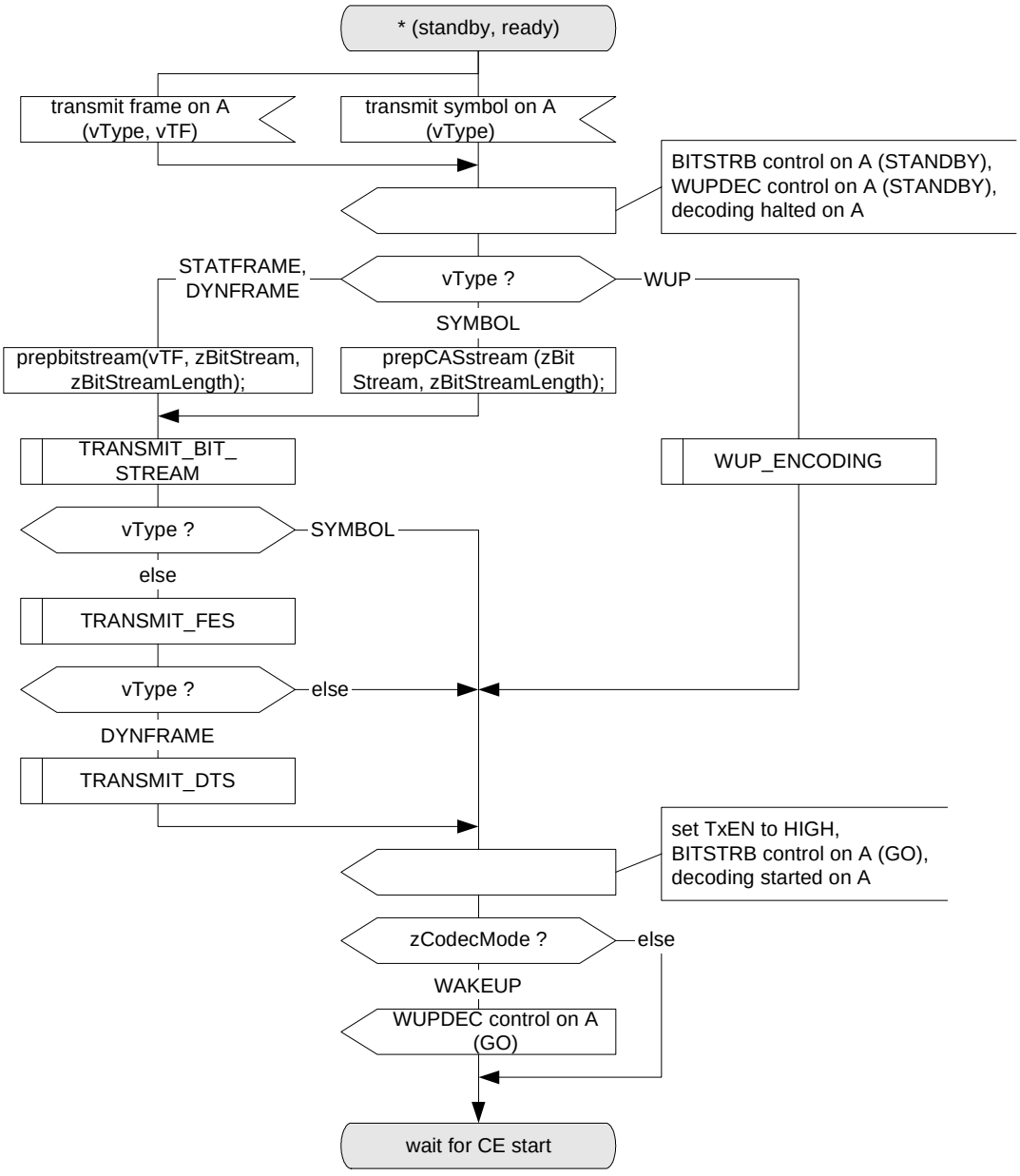

Figure 3-18: Frame encoding [CODEC_A].

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

```
newtype T_Type
      literals STATFRAME, DYNFRAME, SYMBOL, WUP;
endnewtype;
```

Definition 3-5: Formal definition of T_Type.

The transmission of communication elements can be distinguished in four different types:

1. STATFRAME - transmit a frame in the static segment (*transmit frame on A(vType, vTF)*),
2. DYNFRAME - transmit a frame in the dynamic segment (*transmit frame on A(vType, vTF)*),
3. SYMBOL - transmit a symbol in the symbol window (*transmit symbol on A(vType)*),
4. WUP - transmit a wakeup pattern (*transmit symbol on A(vType)*).

The transmitter shall start transmission of a CE by sending the *set TxEN to LOW* signal.

The transmitter shall stop transmission of a CE by sending the *set TxEN to HIGH* signal.

3.3.4 Encoding macros

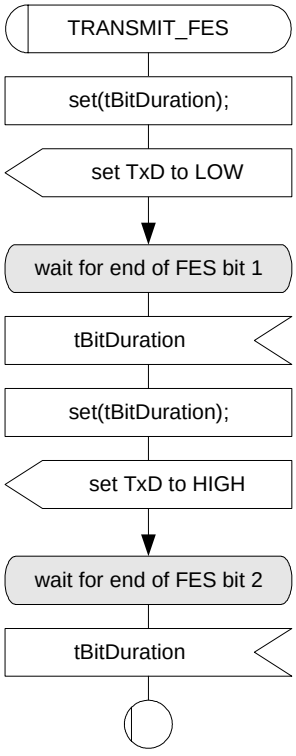

Figure 3-19: Encoding macro TRANSMIT_FES [CODEC_A].

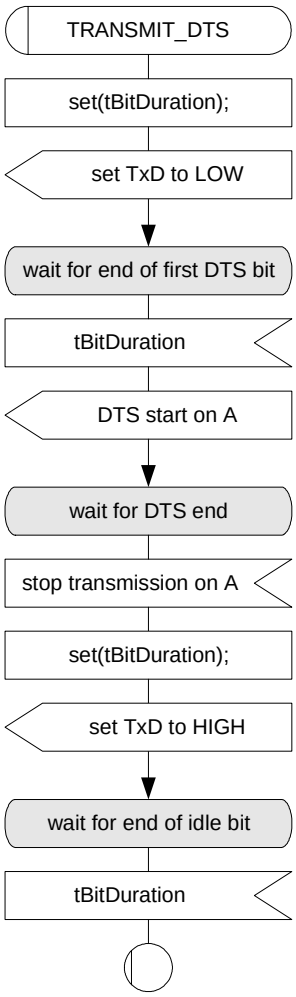

Figure 3-20: Encoding macro TRANSMIT_DTS [CODEC_A].

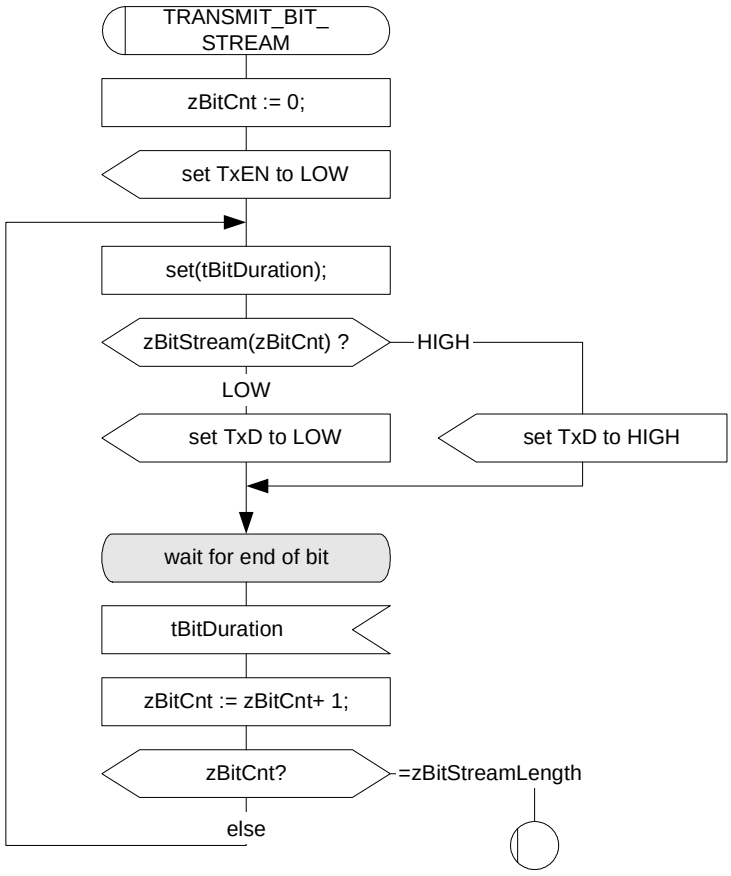

Figure 3-21: Encoding macro TRANSMIT_BIT_STREAM [CODEC_A].

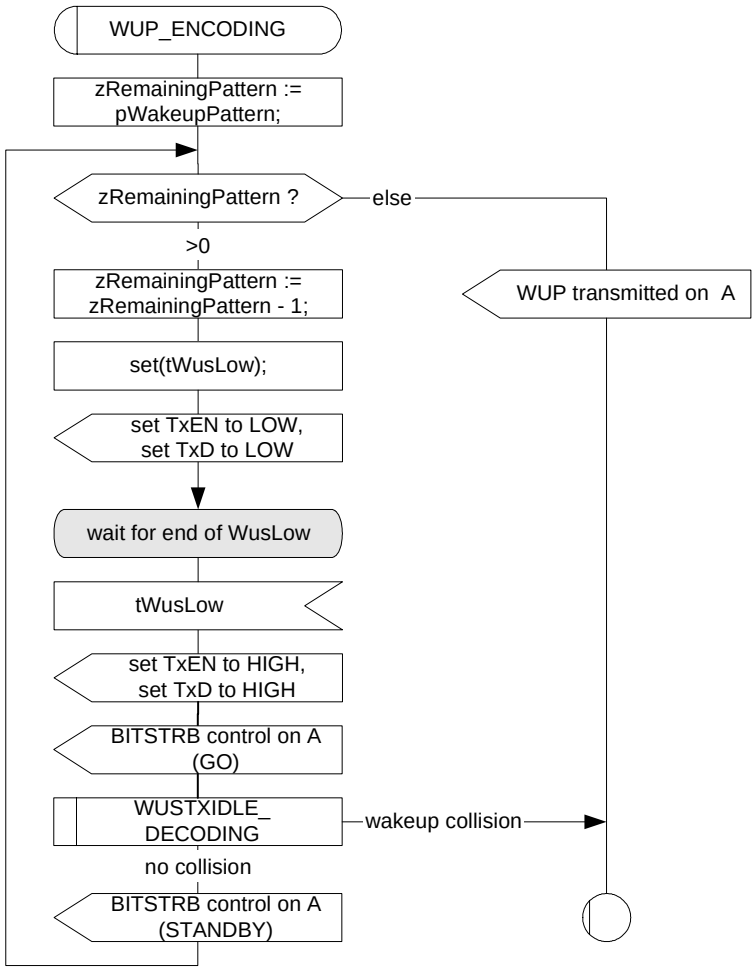

Figure 3-22: Encoding macro WUP_ENCODING [CODEC_A].

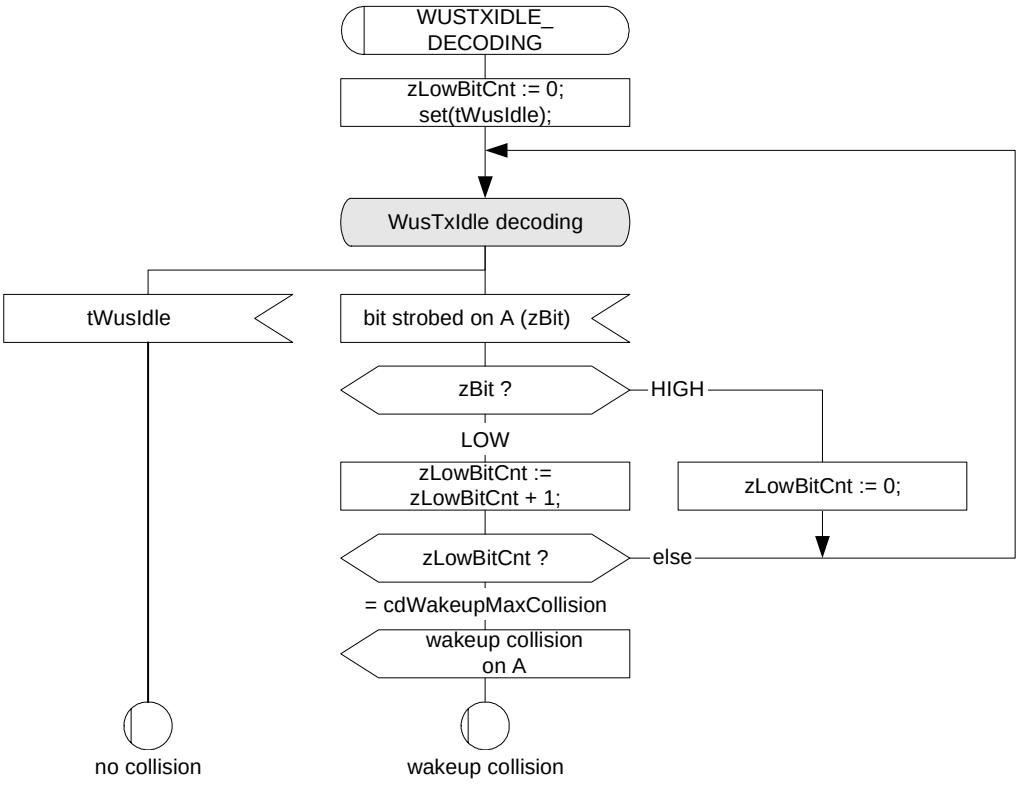

Figure 3-23: Macro WUSTXIDLE_DECODING [CODEC_A].

3.3.5 Decoding behavior

A receiving node shall decode the received data from the *voted RxD on A* signal according to the CODEC_A process.

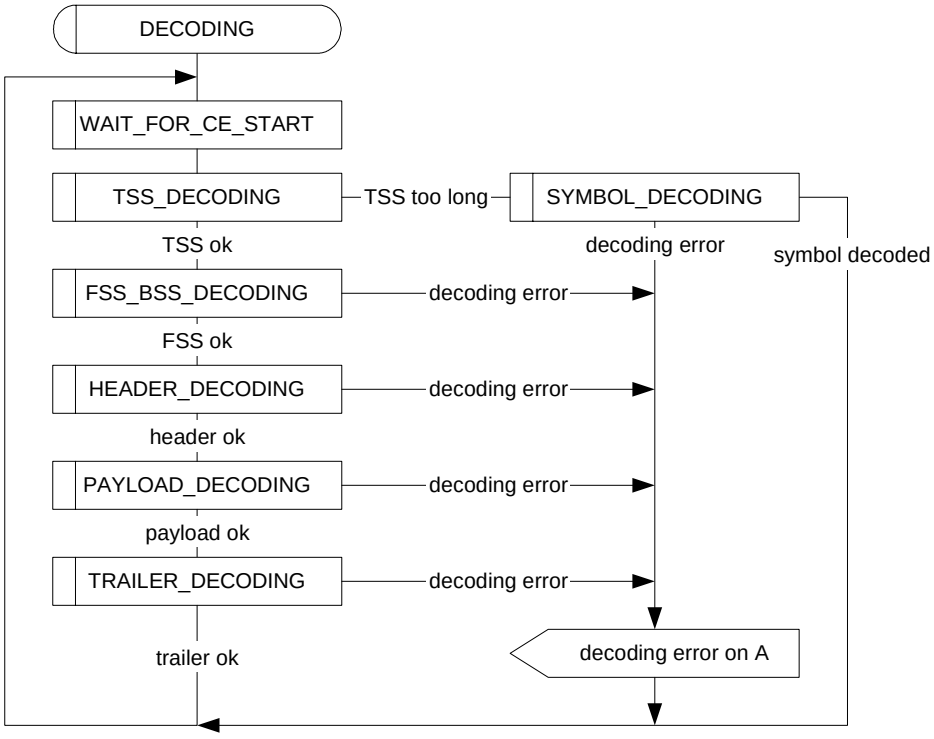

Figure 3-24: Macro DECODING [CODEC_A].

3.3.6 Decoding macros

The following formal definitions are used within the frame decoding macros:[33]

```
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.

[33] Refer to Chapter 6 for further type definitions.

```
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.

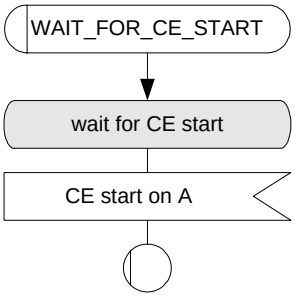

Figure 3-25: Decoding macro WAIT_FOR_CE_START [CODEC_A].

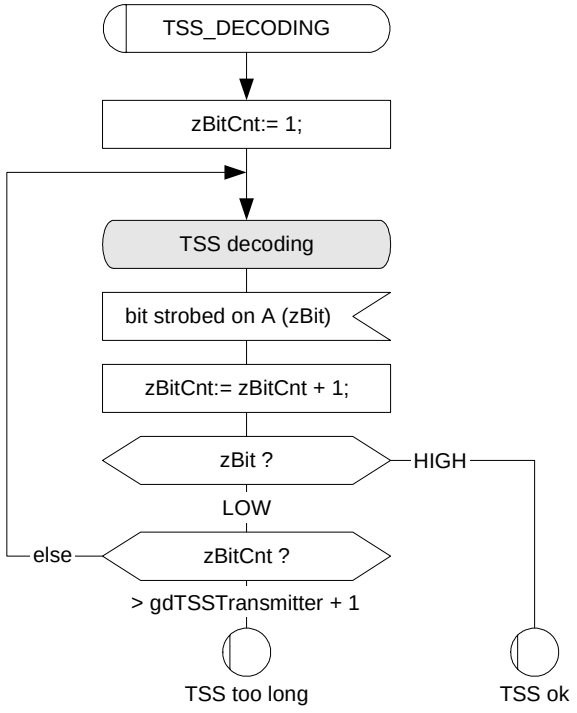

Figure 3-26: Decoding macro TSS_DECODING [CODEC_A].

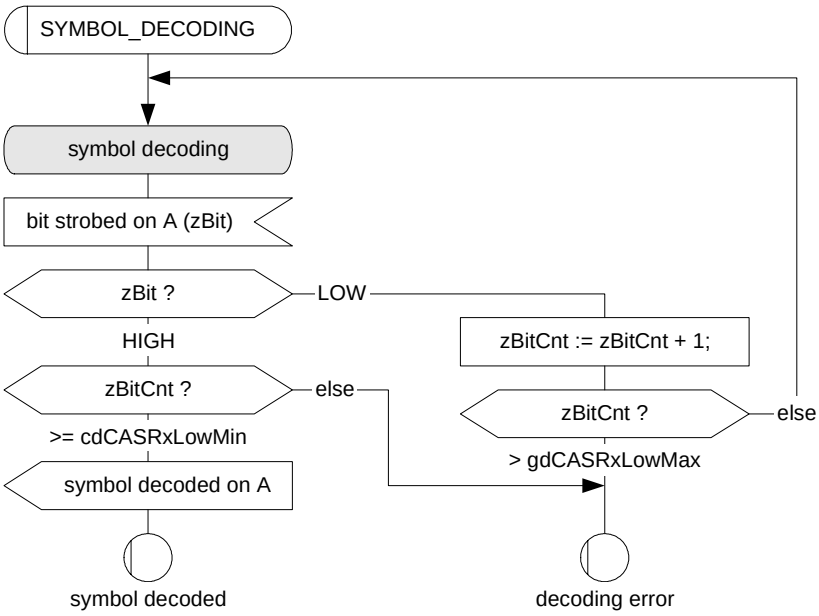

Figure 3-27: Decoding macro SYMBOL_DECODING [CODEC_A].

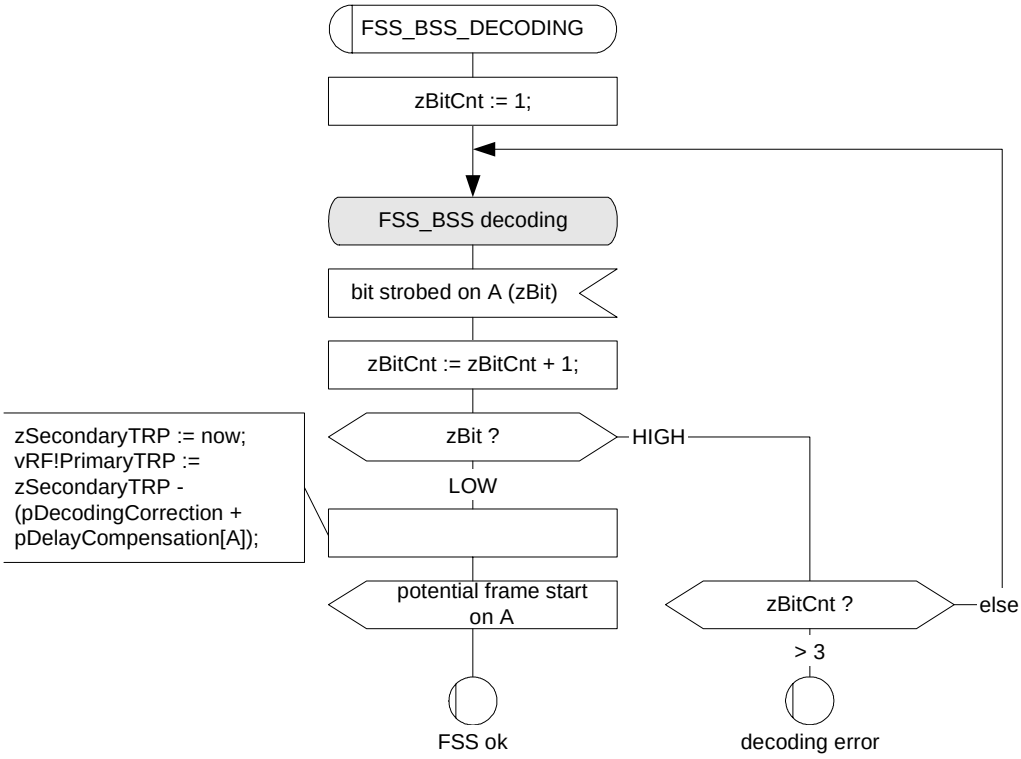

Figure 3-28: Decoding macro FSS_BSS_DECODING [CODEC_A].[34]

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

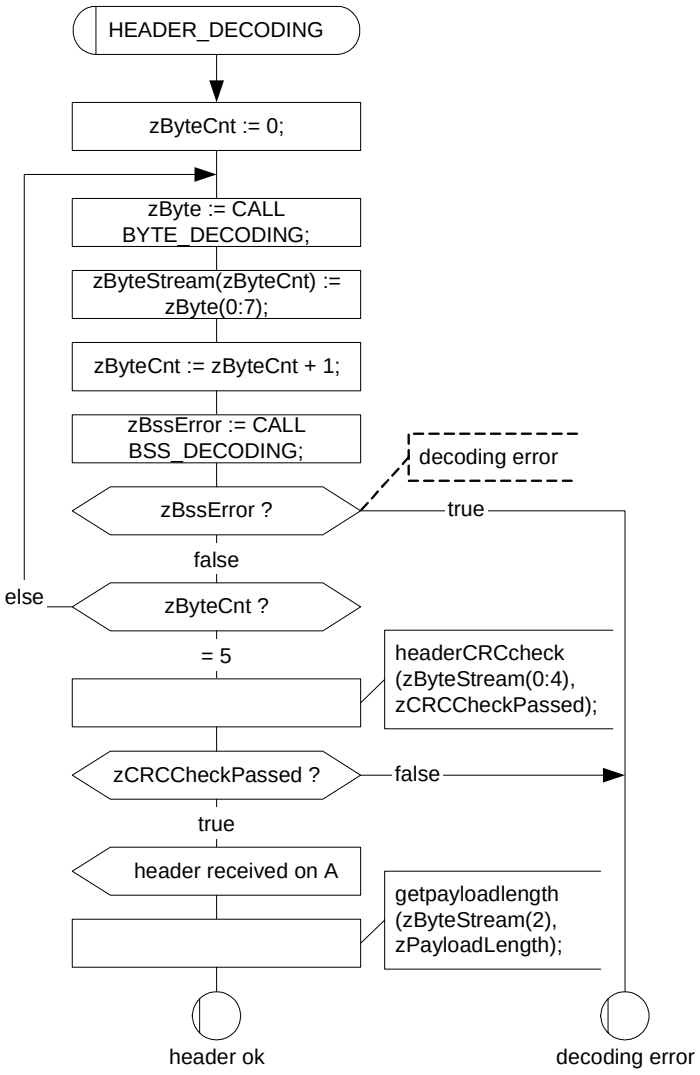

Figure 3-29: Decoding macro HEADER_DECODING [CODEC_A].

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 **getpayloadlength** 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 3-33 and Figure 3-34.

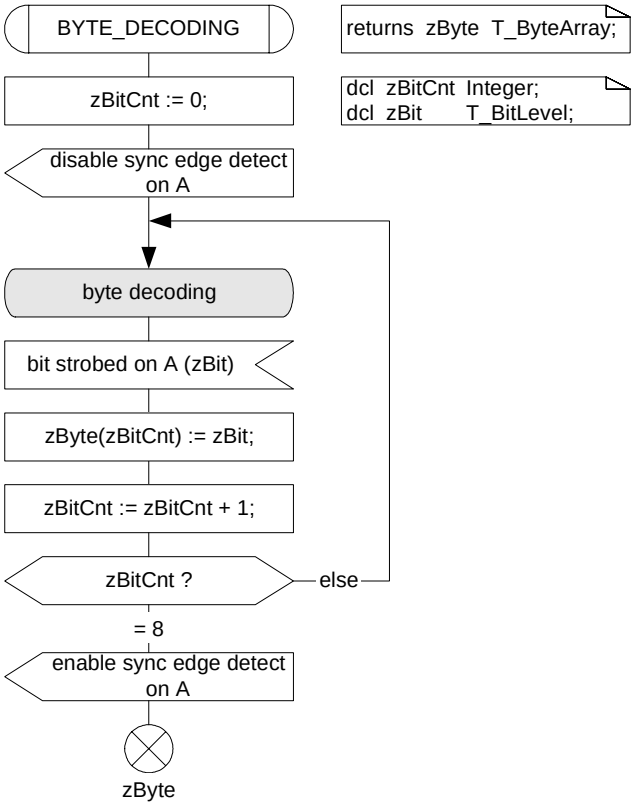

Figure 3-30: Procedure for Byte Decoding [CODEC_A].

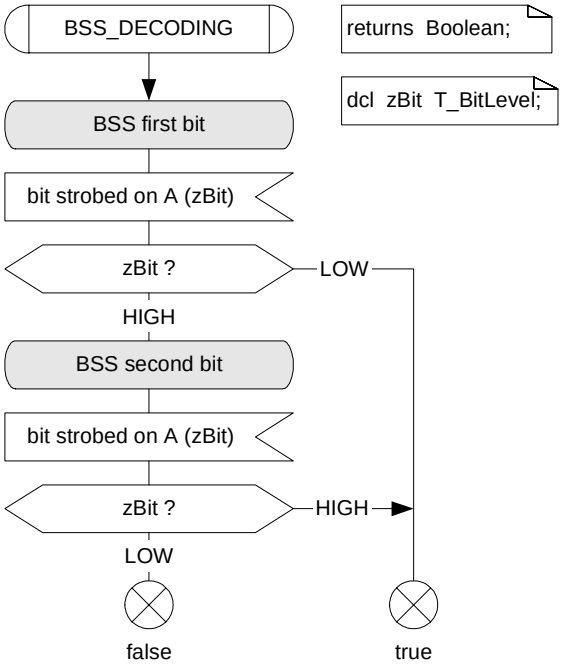

Figure 3-31: Procedure for BSS Decoding [CODEC_A].

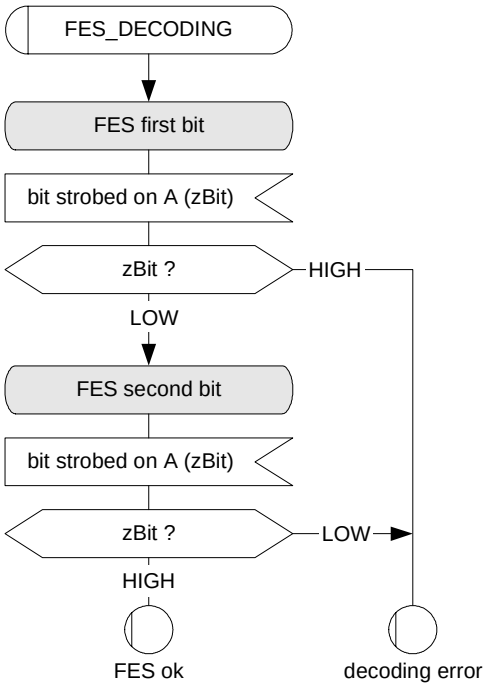

Figure 3-32: Decoding macro FES_DECODING [CODEC_A].

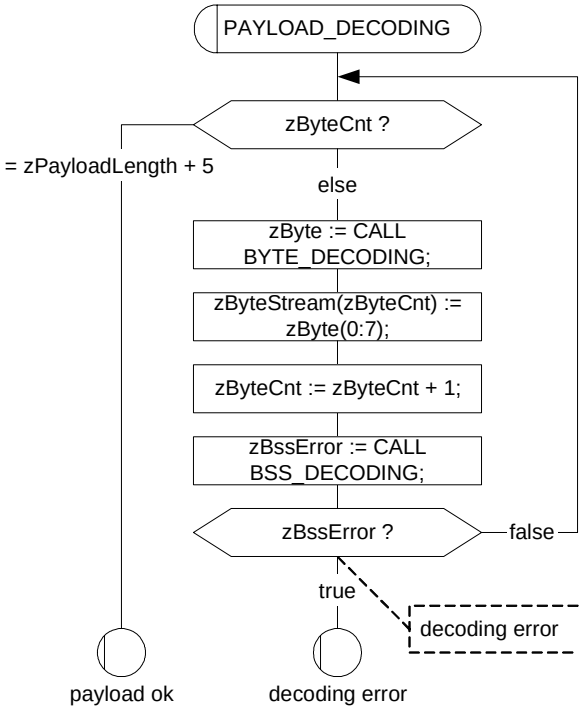

Figure 3-33: Decoding macro PAYLOAD_DECODING [CODEC_A].

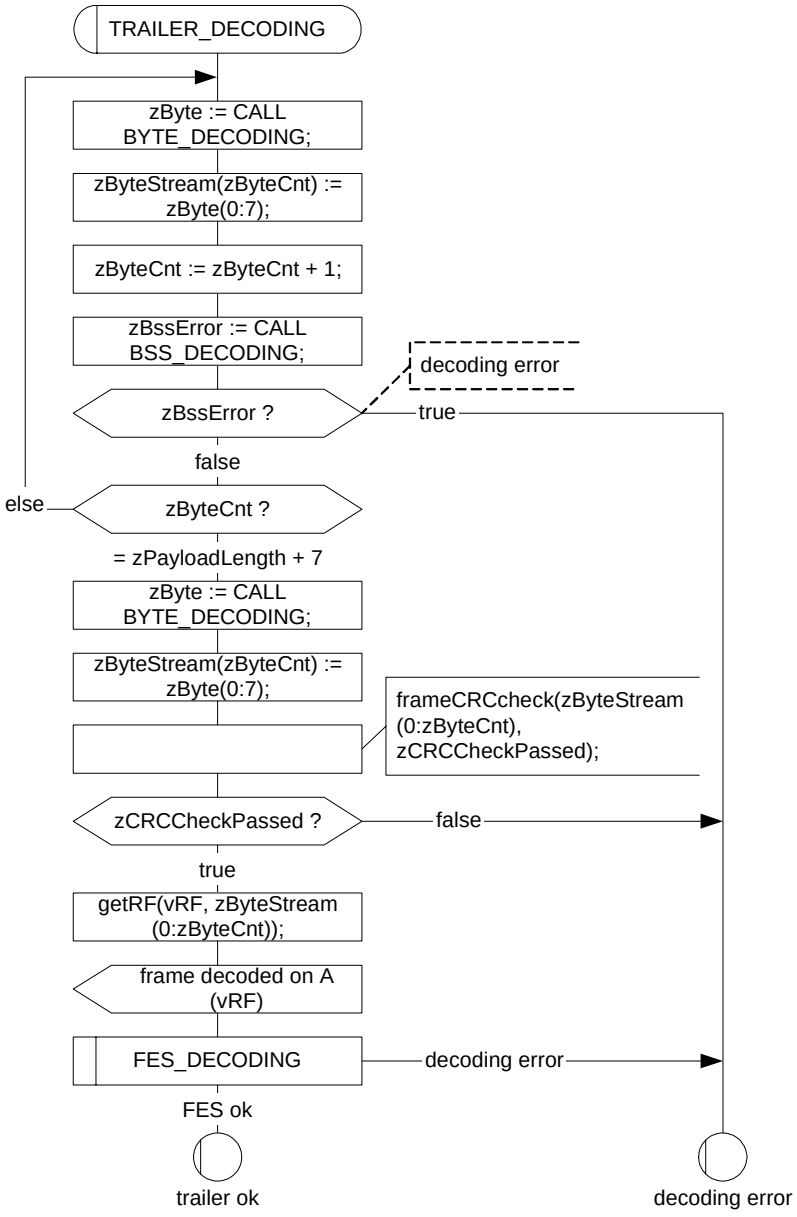

Figure 3-34: Decoding macro TRAILER_DECODING [CODEC_A].

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 3-34 extracts decoded header and payload data from *zByteStream* and returns it via the structure variable *vRF*.

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 3-11 shows the formal definition of the BITSTRB operating modes:

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

Definition 3-11: Formal definition of T_StrbMode.

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

1. In the STANDBY mode bit strobing is effectively halted.
2. In the GO mode the bit strobing process shall be executed.

At the transition from the state *CODEC:standby* to the state *CODEC:ready* 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

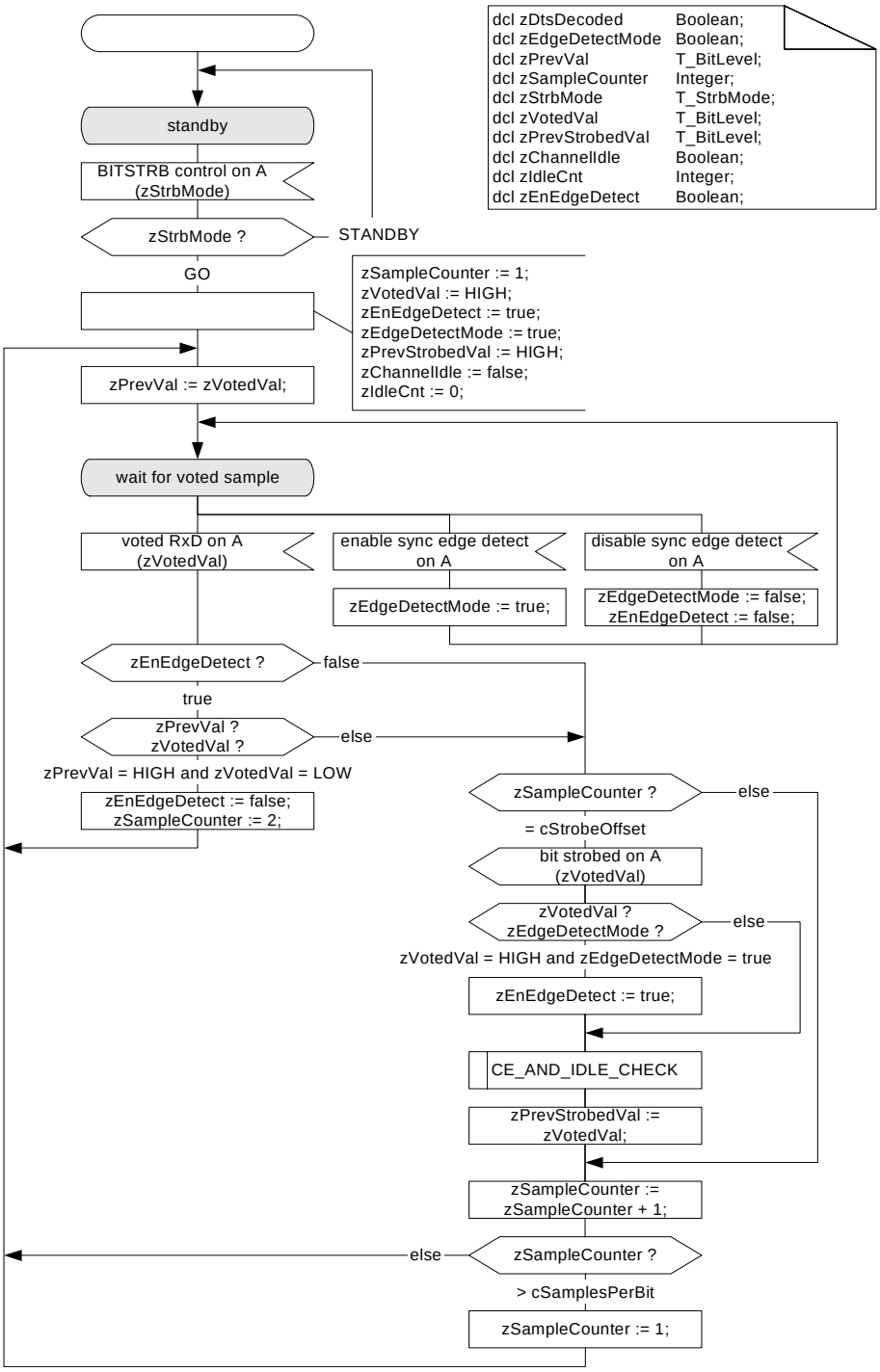

Figure 3-35: BITSTRB process [BITSTRB_A].

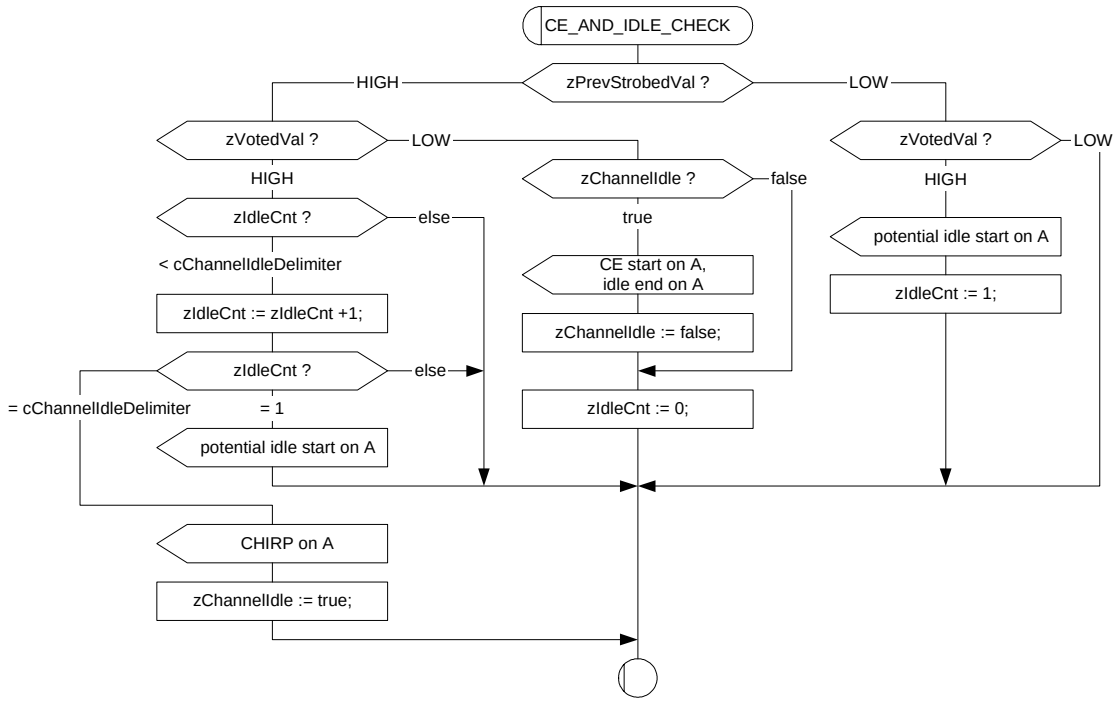

Figure 3-36: BITSTRB process macro CE_AND_IDLE_CHECK [BITSTRB_A].

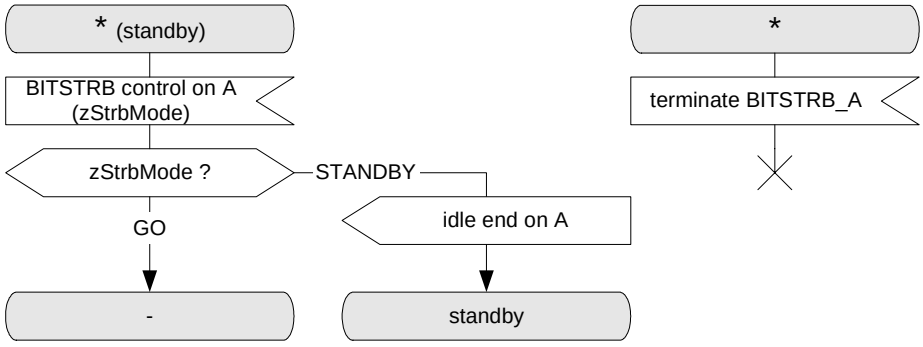

Figure 3-37: BITSTRB process control and process termination [BITSTRB_A].

3.5 Wakeup pattern decoding process

3.5.1 Operating modes

The wakeup pattern decoding process WUPDEC distinguishes between two modes. Definition 3-12 shows the formal definition of the WUPDEC operating modes:

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

Definition 3-12: 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 receiving node shall decode wakeup patterns.

3.5.2 Wakeup decoding process behavior

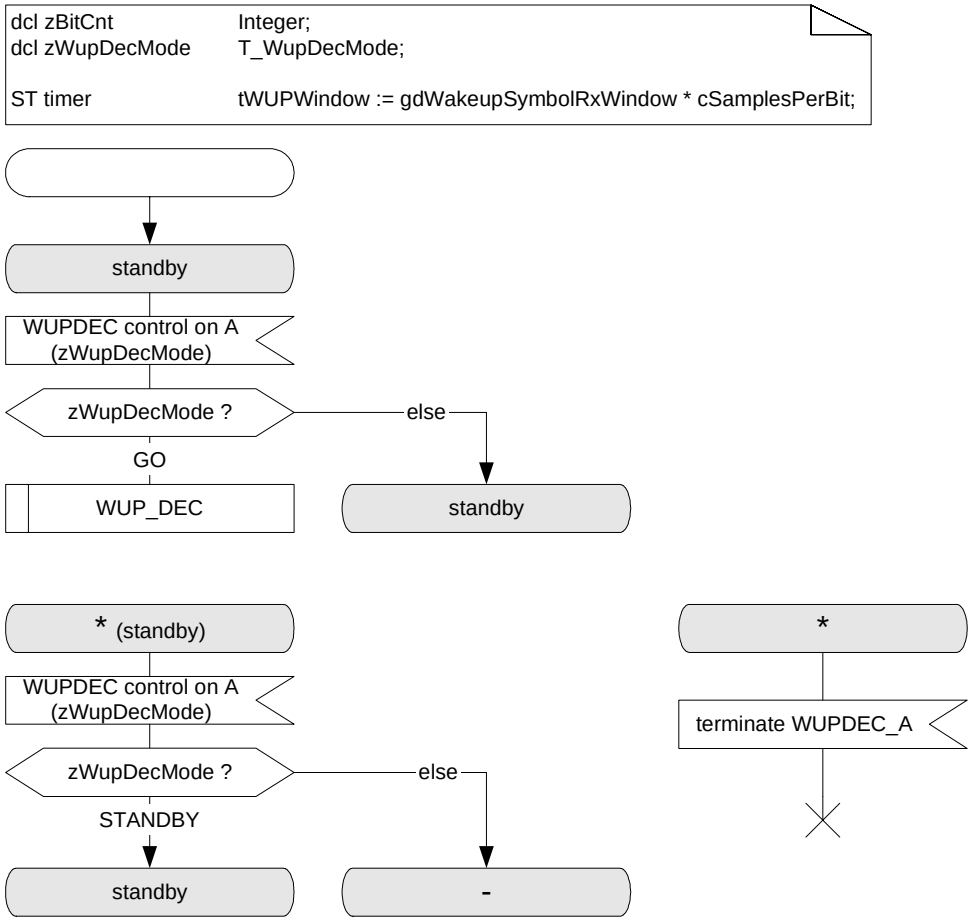

Figure 3-38: Control of the wakeup pattern detection process and its termination [WUPDEC_A].

3.5.3 Wakeup decoding macros

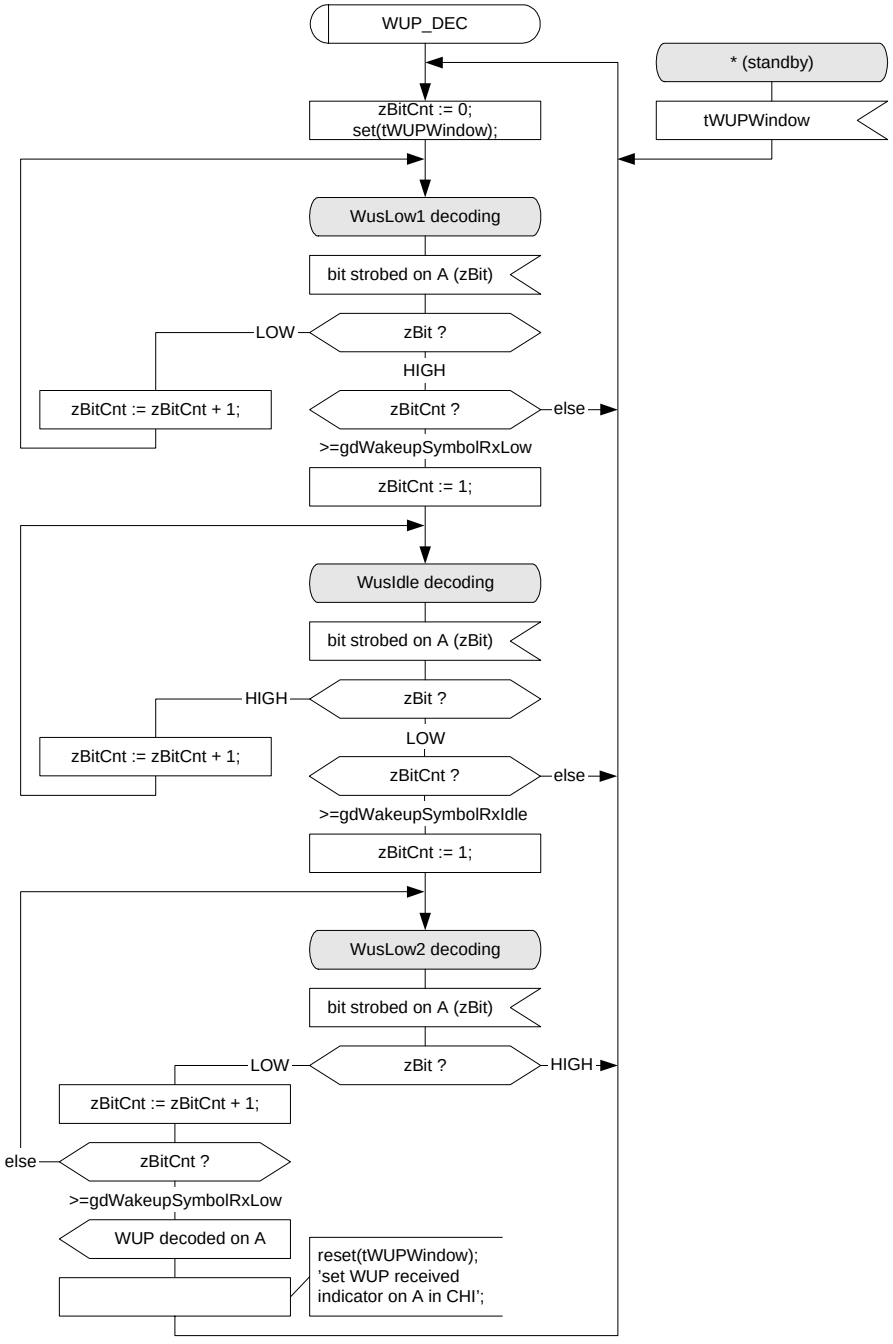

Figure 3-39: Wakeup pattern decoding Macro [WUPDEC_A].

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.

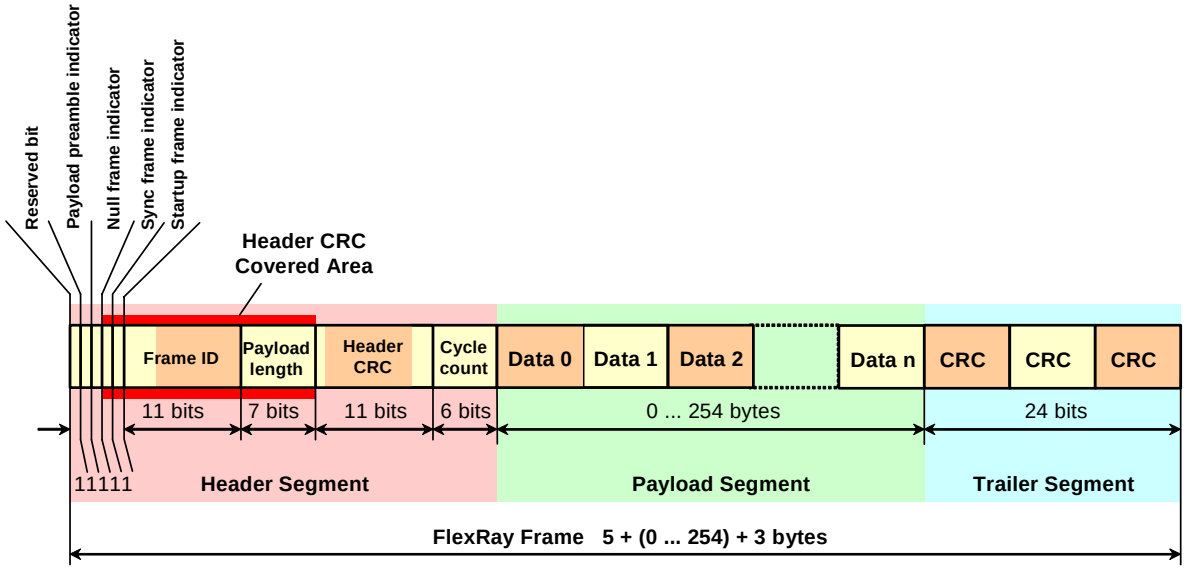

Figure 4-1: FlexRay frame format.

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.[35]

```
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[36]:

- 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.[37] 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.[38]

- If the null frame indicator is set to zero then the payload segment contains no valid data. All bytes in the payload section are 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.[39]

[35] 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).

[36] If the null frame indicator is set to zero the payload preamble indicator is irrelevant because the payload contains no usable data.

[37] 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.

[38] 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).

- 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 (see Chapter 7). Only *coldstart nodes* are allow 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.

Since the startup frame indicator can only be set to one in sync frames, every coldstart node can transmit exactly one frame per communication cycle and channel with the startup frame indicator set to one.

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.[40]

```
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[41]. The frame ID 0 is an invalid frame ID[42].

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.

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

[40] 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.

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

[42] 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.

```
syntype T_FrameID = Integer
      constants 0 : 2047[43]
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).[44]

The payload length ranges from 0 to *cPayloadLengthMax*, 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 *gPayload-LengthStatic*.

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).[45] 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[46] 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.

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

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

[45] 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).

[46] 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].

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.

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 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.[47] 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.

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 optionally 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[48]. 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.[49]

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 (optional)

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.[50]

[47] The length of the payload segment indicated by the Length field of the T_Header structure corresponds to the number of bytes that are sent on the communication channel. It does not necessarily correspond to the number of bytes used by the application in the payload segment. The data provided by the application may be shorter than the payload segment. A padding function in the communication controller fills the "missing" bytes if the configured transmit buffer is smaller than the configured payload length.

[48] 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.

[49] 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.

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

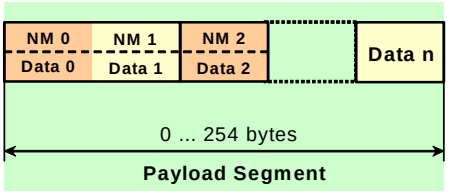

Figure 4-2: Payload segment of frames transmitted in the static segment.

4.3.2 Message ID (optional, 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 may 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 communication controller has no knowledge about the message ID and no mechanism inside the communication controller 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 the Chapter 9.
- The optional presence of message IDs 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 MessageID shall be placed in the next payload bits in order of decreasing significance.

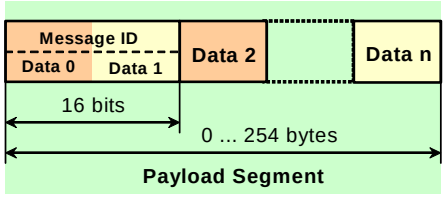

Figure 4-3: Payload segment of frames transmitted in the dynamic segment.

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.[51]

The CRC is computed using the same generator polynomial on both channels. The CRC polynomial[52] shall be

[51] 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[53]:

- 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 3-29 and Figure 3-34). 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[54] 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)
```

[52] 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].

[53] 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).

[54] 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[55].

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:

[55] 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 header 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.

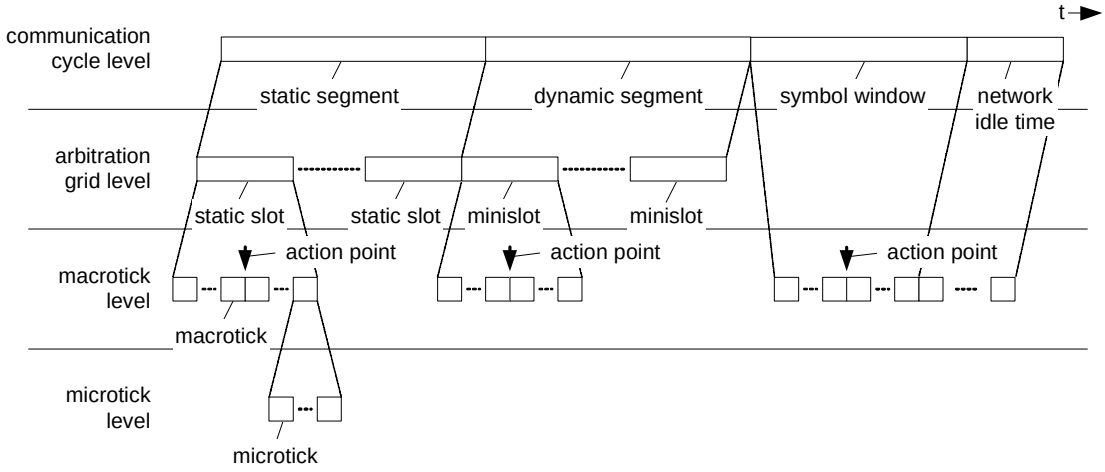

Figure 5-1: Timing hierarchy within the communication cycle.

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 *cCycleCountMax*.

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.

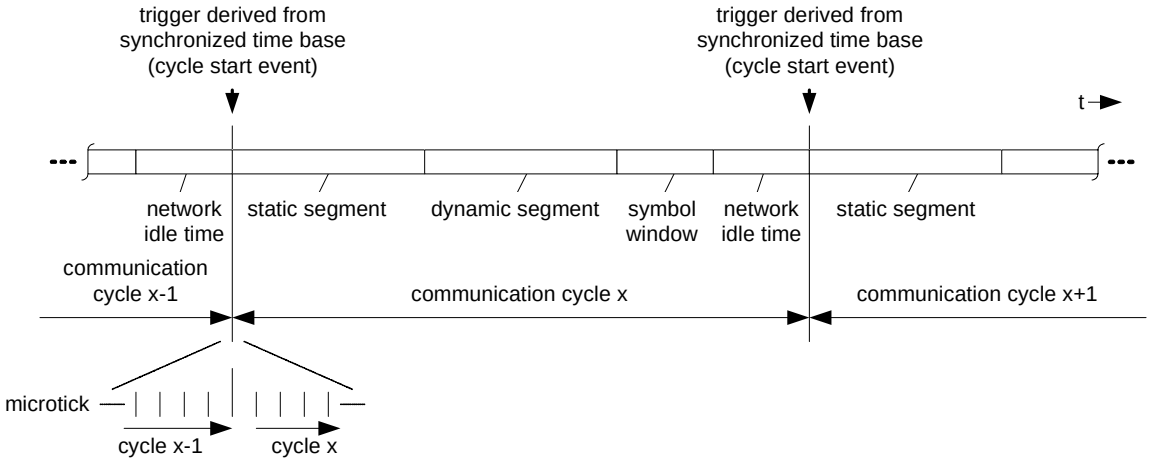

Figure 5-2: Time base triggered communication cycle.

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 process for channel A*. The node shall contain an equivalent media access 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. Sync frames shall be transmitted on all connected channels.
2. Non-sync frames may be transmitted on either channel, or both.
3. Only one node shall transmit a given frame ID on a given channel.[56]
4. If the cluster is configured for single slot mode, all non-sync nodes shall designate a frame as the single slot frame.

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[57]. In slot 3 no frame is transmitted on either channel.

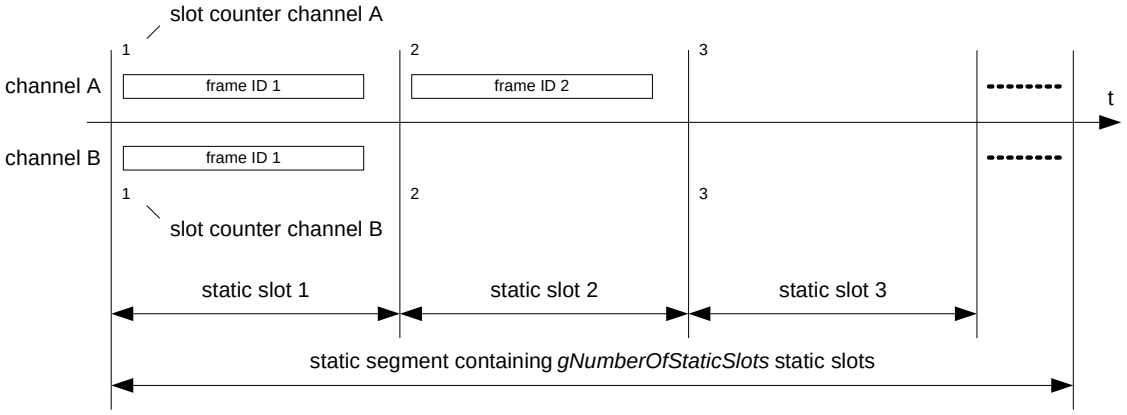

Figure 5-3: Structure of the static segment.

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

[56] This requirement applies to the entire operation of the cluster, as opposed to only a single cycle. For example, it is not acceptable to configure a cluster such that different nodes transmit in the same slot/channel combination in different cycles.

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

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. Appropriate configuration of the static slot length *gdStaticSlot* must assure that the frame and the channel idle delimiter and any potential safety margin fit within the static slot under worst-case assumptions. Details are given in Appendix B.

For any given node one static slot (as defined in *pKeySlotId*) may be assigned to contain a sync frame (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.

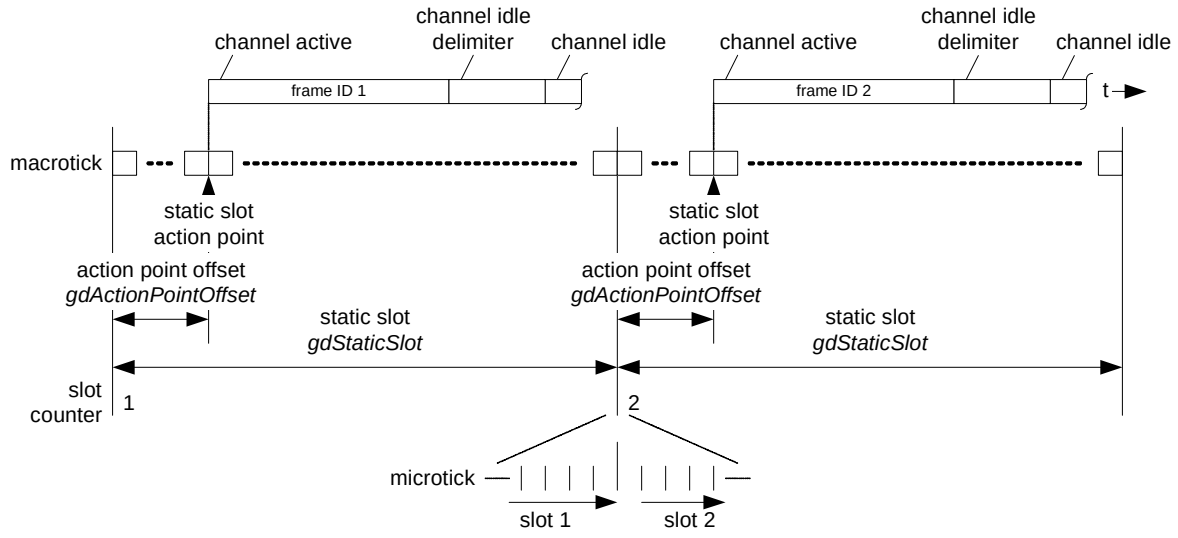

Figure 5-4: Timing within the static segment.

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. The formula for determining the parameter *gdActionPointOffset* and the corresponding constraints are specified in Appendix B.

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.

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.

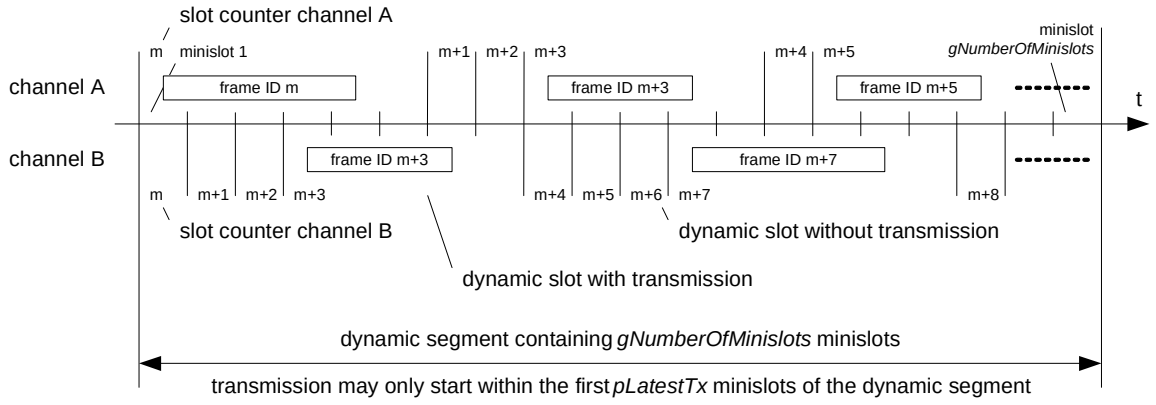

Figure 5-5: Structure of the dynamic segment.

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 communication 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 multiple minislots if communication takes place on the channel.

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.

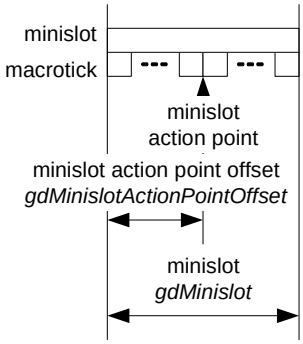

Figure 5-6: Timing within a minislot.

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 distinguishes between the *dynamic slot transmission phase* and the *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* concludes the dynamic slot. The dynamic slot idle phase is defined as a communication-free phase that succeeds the transmission phase in each dynamic slot. It is required to account for the communication channel idle detection latency and to process the frame by the receiving nodes.

Figure 5-7 depicts the detailed timing within the dynamic segment.

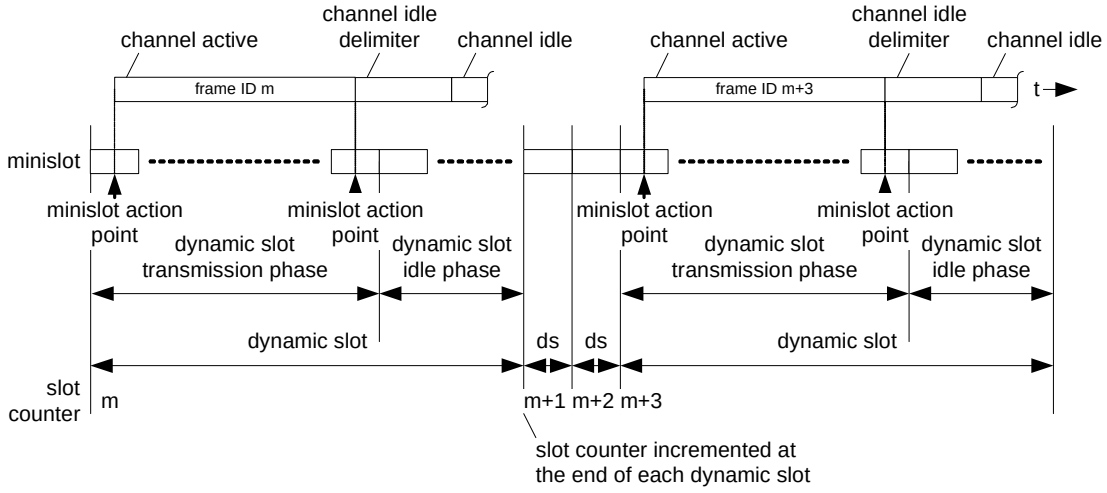

Figure 5-7: Timing within the dynamic segment.

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.[58]

The two cases are illustrated in Figure 5-8.

[58] 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.

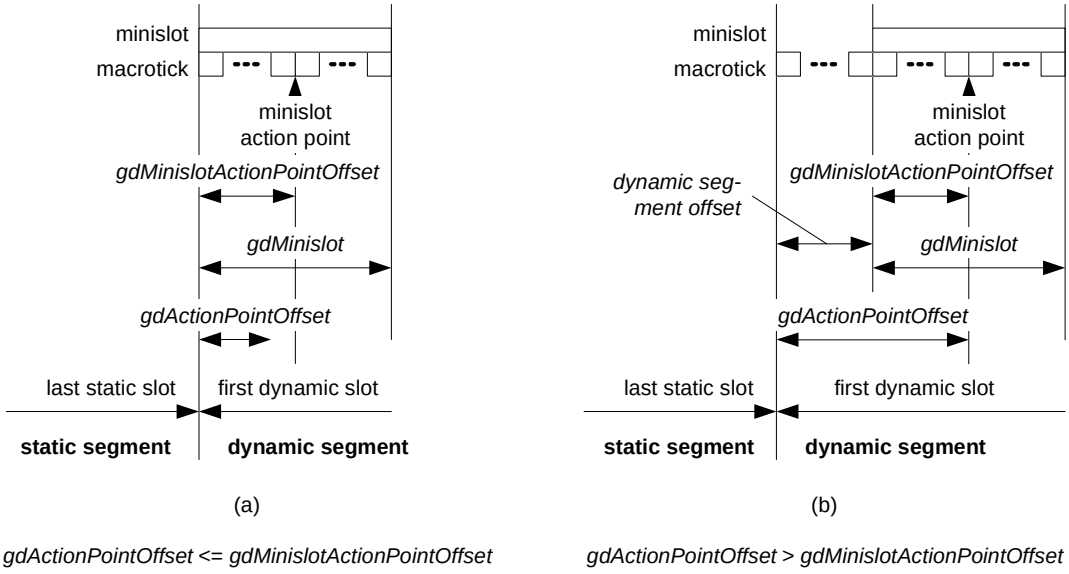

Figure 5-8: Timing at the boundary between the static and dynamic segments.

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

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.

5.1.5 Symbol window

Within the symbol window a single symbol 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. As illustrated in Figure 5-9 the symbol window resembles a static slot.

The number of macroticks per symbol window *gdSymbolWindow* is a global constant for a given cluster. Configuration of the symbol window length *gdSymbolWindow* is defined in Appendix B.

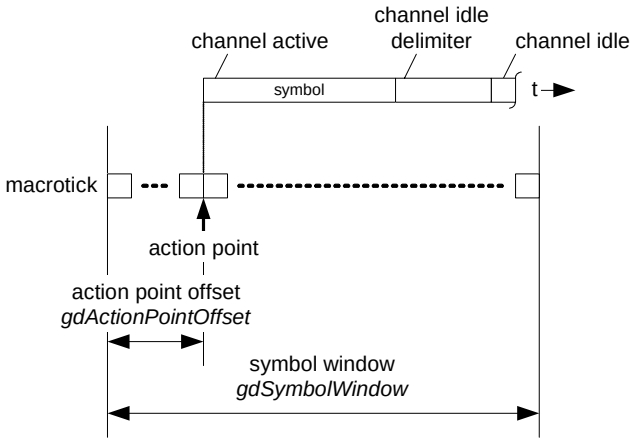

Figure 5-9: Timing within the symbol window.

The symbol window contains an action point that is offset from the start of the slot by *gdActionPointOffset* macroticks. The number of macroticks within the action point offset is identical to the number of macroticks used in the static slot.

A symbol transmission starts at the action point 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 the node performs implementation specific 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.

Constraints on the duration of the network idle time are specified in Appendix B.

5.2 Description

The relationship between the Media Access Control processes and the other protocol processes is depicted in Figure 5-10[59].

[59] 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.

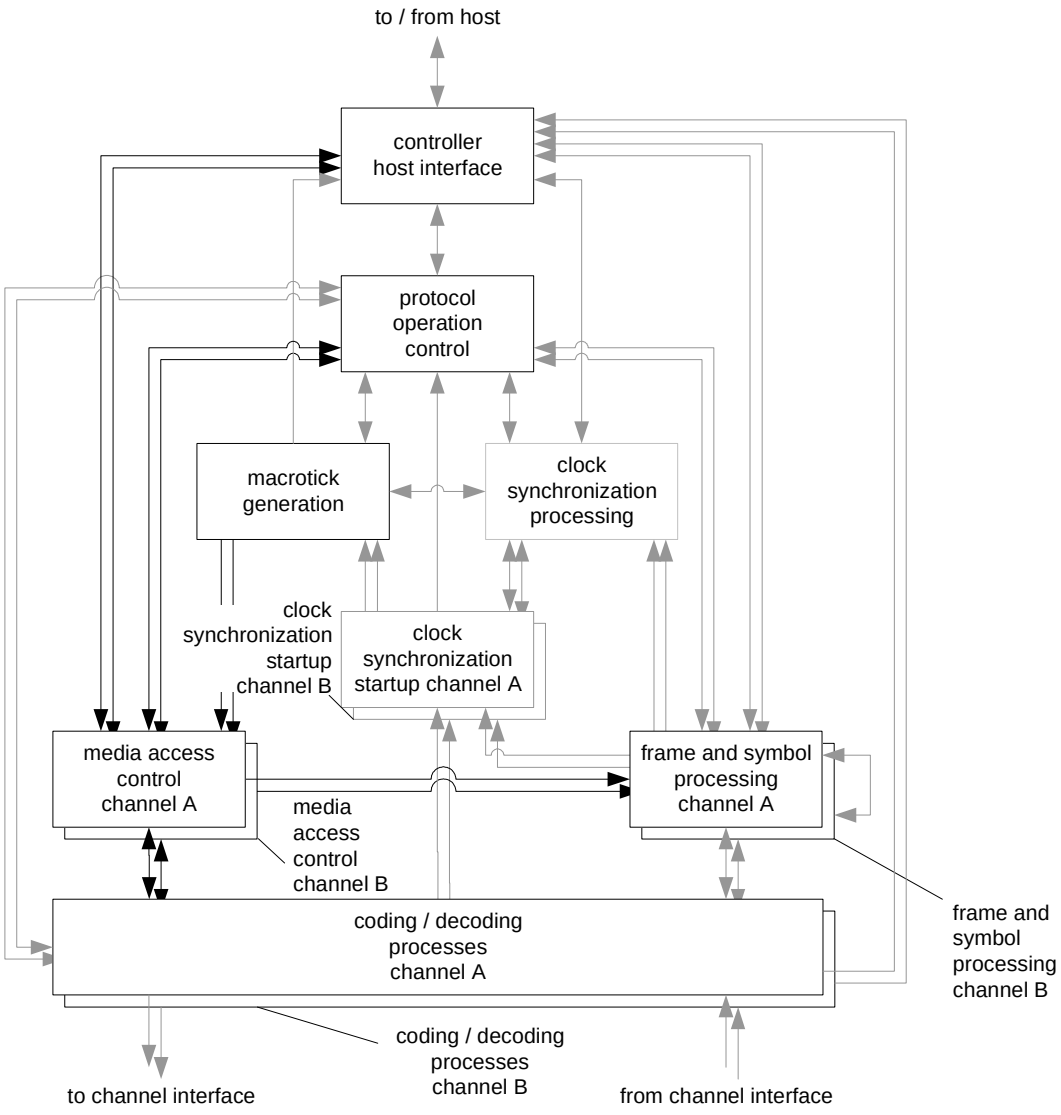

Figure 5-10: Media access control context.

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 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 if the node is configured to send a startup frame.
5. In the SINGLESLOT mode transmissions are restricted to the transmission of one sync frame per cycle on each configured channel if the node is configured to send a sync frame, or to the transmission of a specified single-slot frame per cycle on each configured channel if the node is configured to support the single-slot mode.
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, SINGLESLOT, 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.

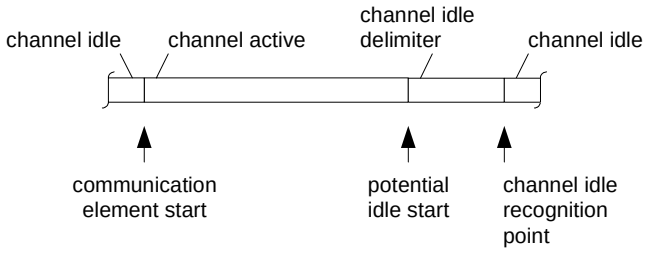

Figure 5-11: Reception-related events for MAC.

For communication channel A the reception-relevant events are:

1. communication element start on channel A (signal *CE start on A*),
2. potential idle start on channel A (signal *potential idle start on A*), and,
3. channel idle recognition point detected on channel A (signal *CHIRP on A*).

For communication channel B the reception-relevant events are:

1. communication element start on channel B (signal *CE start on B*),
2. potential idle start on channel B (signal *potential idle start on B*), and,
3. channel idle recognition point detected on channel B (signal *CHIRP on B*).

5.2.2.2 Transmission-related events

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

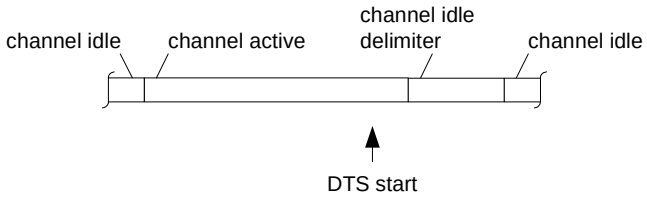

Figure 5-12: Transmission-related events for MAC.

For communication channel A the transmission-relevant event is 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 event is 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 end of the dynamic segment offset* state,
7. a *MAC:wait for the AP transmission start* state,
8. a *MAC:wait for the DTS start* state,
9. a *MAC:wait for the AP transmission end* state,
10. a *MAC:wait for the end of the dynamic slot tx* state,
11. a *MAC:wait for the end of the dynamic segment* state,
12. a *MAC:wait for the end of the minislot* state,
13. a *MAC:wait for the end of the reception* state,
14. a *MAC:wait for the end of the dynamic slot rx* state,
15. a *MAC:wait for the symbol window action point state*, and,
16. a *MAC:wait for the end of the symbol window* state.

The node shall access all media as specified in the FlexRay Physical Layer Specification according to the media access control process.

5.3.1 Initialization and state *MAC:standby*

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

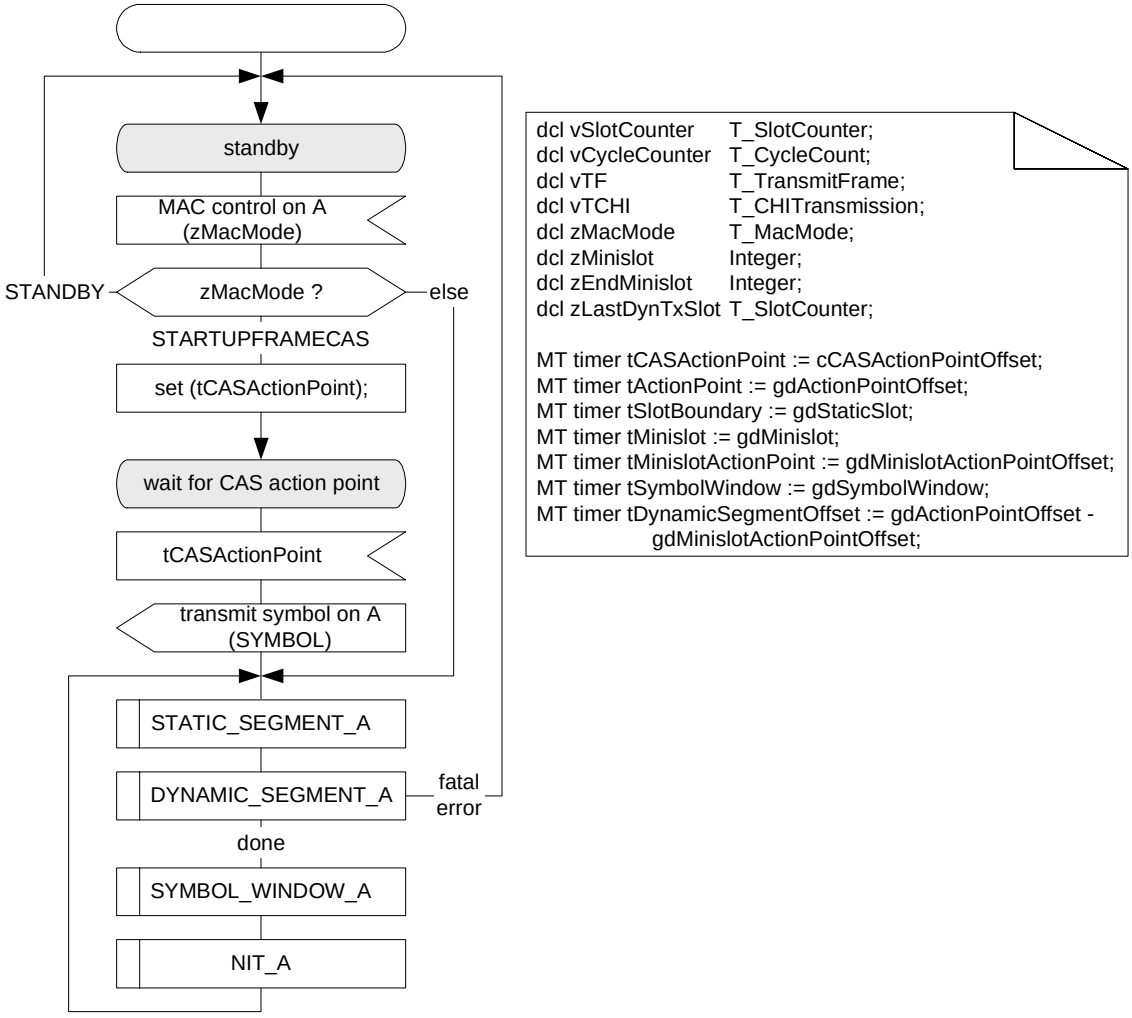

Figure 5-13: Media access process [MAC_A].

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

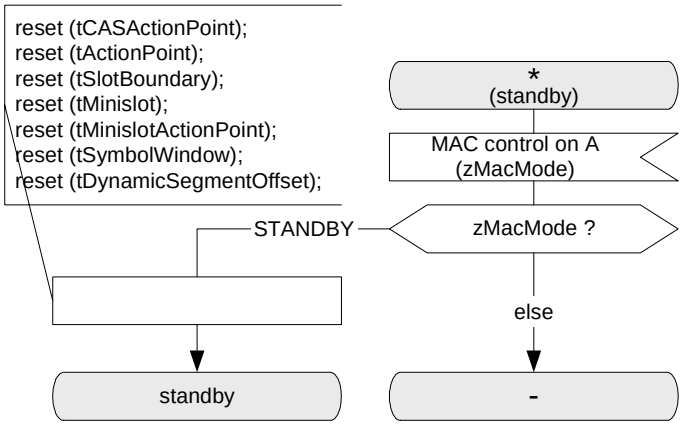

Figure 5-14: Media access control [MAC_A].

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.

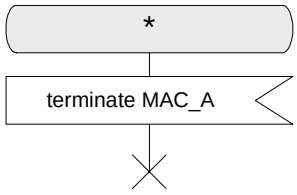

Figure 5-15: Termination of the MAC process [MAC_A].

5.3.2 Static segment related states

5.3.2.1 State machine for the static segment media access control

Figure 5-16 gives an overview of the media access states in the static segment and how they interrelate.

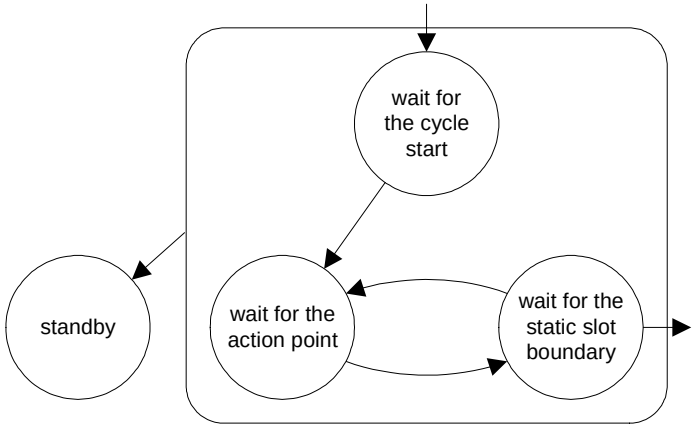

Figure 5-16: Media access in the static segment [MAC_A].

The node shall perform media access in the static segment according to Figure 5-17 for channel A.

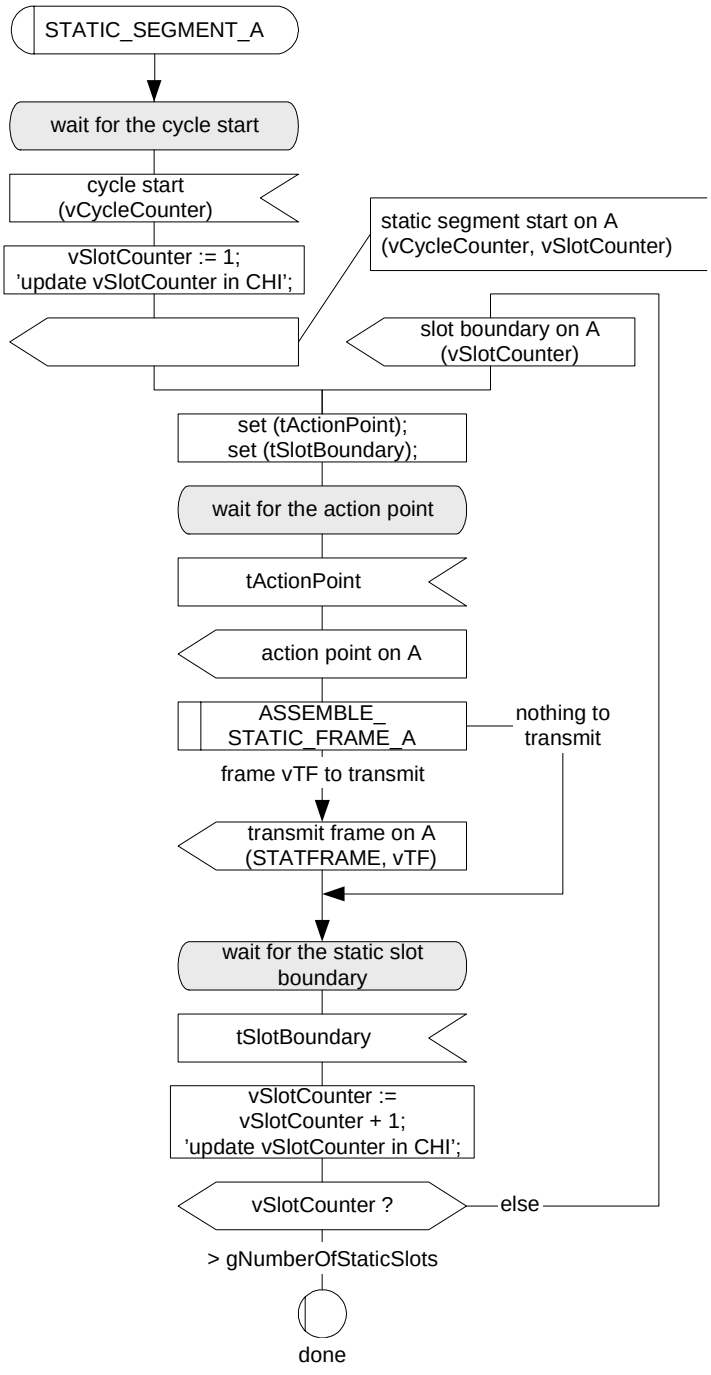

Figure 5-17: Media access in the static segment [MAC_A].

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 5-2.

```
newtype T_TransmitFrame
struct
```

```
      Header      T_Header;
      Payload     T_Payload;
endnewtype;
```

Definition 5-2: Formal definition of T_TransmitFrame.

Definition 5-3 defines *T_SlotCounter*.

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

Definition 5-3: 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_A. 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 both channels if the communication slot is an assigned startup 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 5-4.

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

Definition 5-4: Formal definition of T_CHITransmission.

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

Definition 5-5: 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 *pKeySlotId* equals *vSlotCounter* 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. *vTCHI!Length* number of two-byte payload words shall be copied from *vTCHI!Message* to *vTF!Payload*, and

d. any remaining payload data words shall be set to the padding pattern 0x00

else if the host has no data available (*vTCHI!TxMessageAvailable* set to false) then

e. the null frame indicator shall be set to zero, and

f. the payload preamble indicator shall be set to zero, and

g. all payload data shall be set to the padding pattern 0x00.

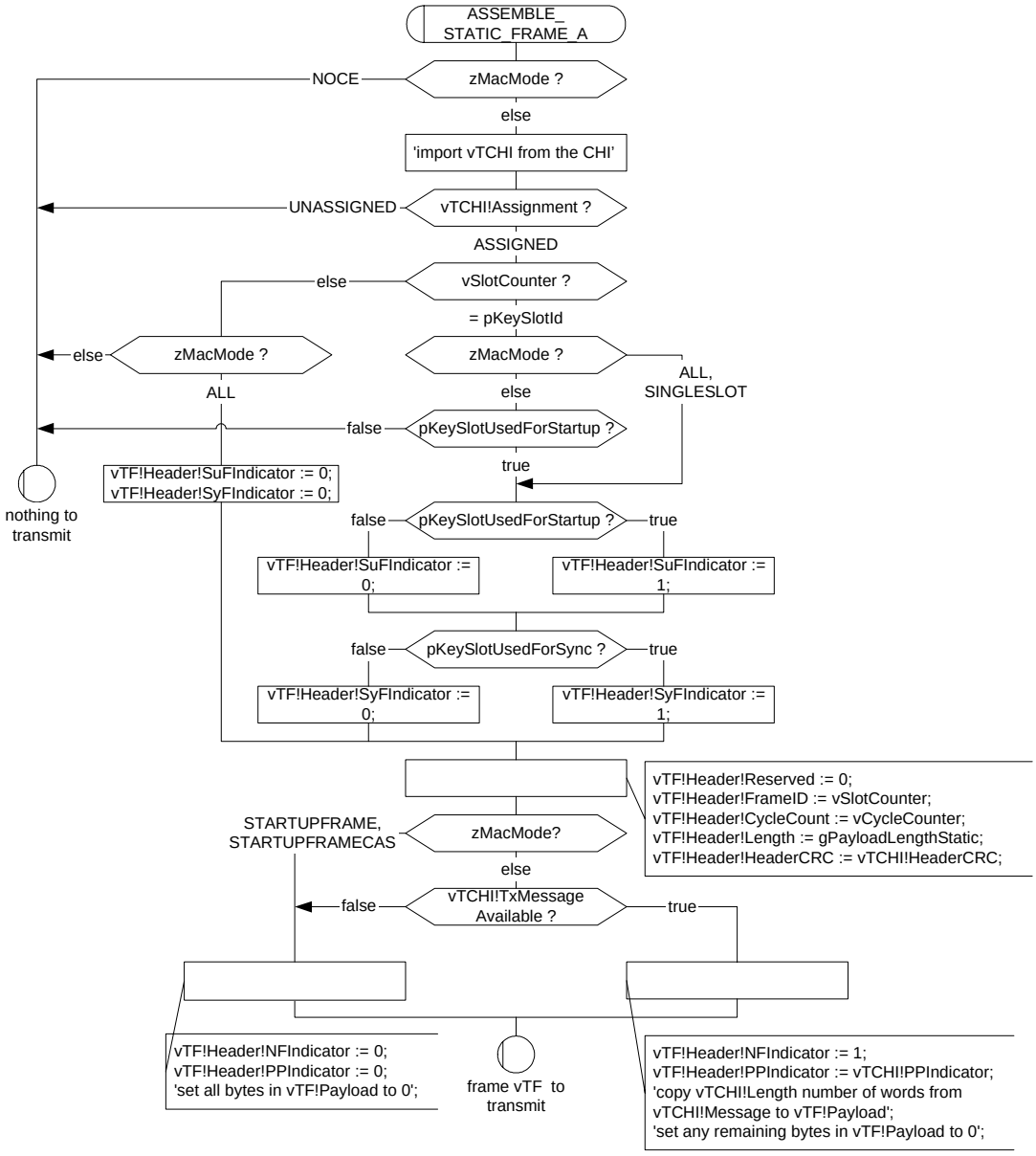

Figure 5-18: Frame assembly in the static segment [MAC_A].

5.3.3 Dynamic segment related states

5.3.3.1 State machine for the dynamic segment media access control

Figure 5-19 gives an overview of the media access states in the dynamic segment and how they interrelate.

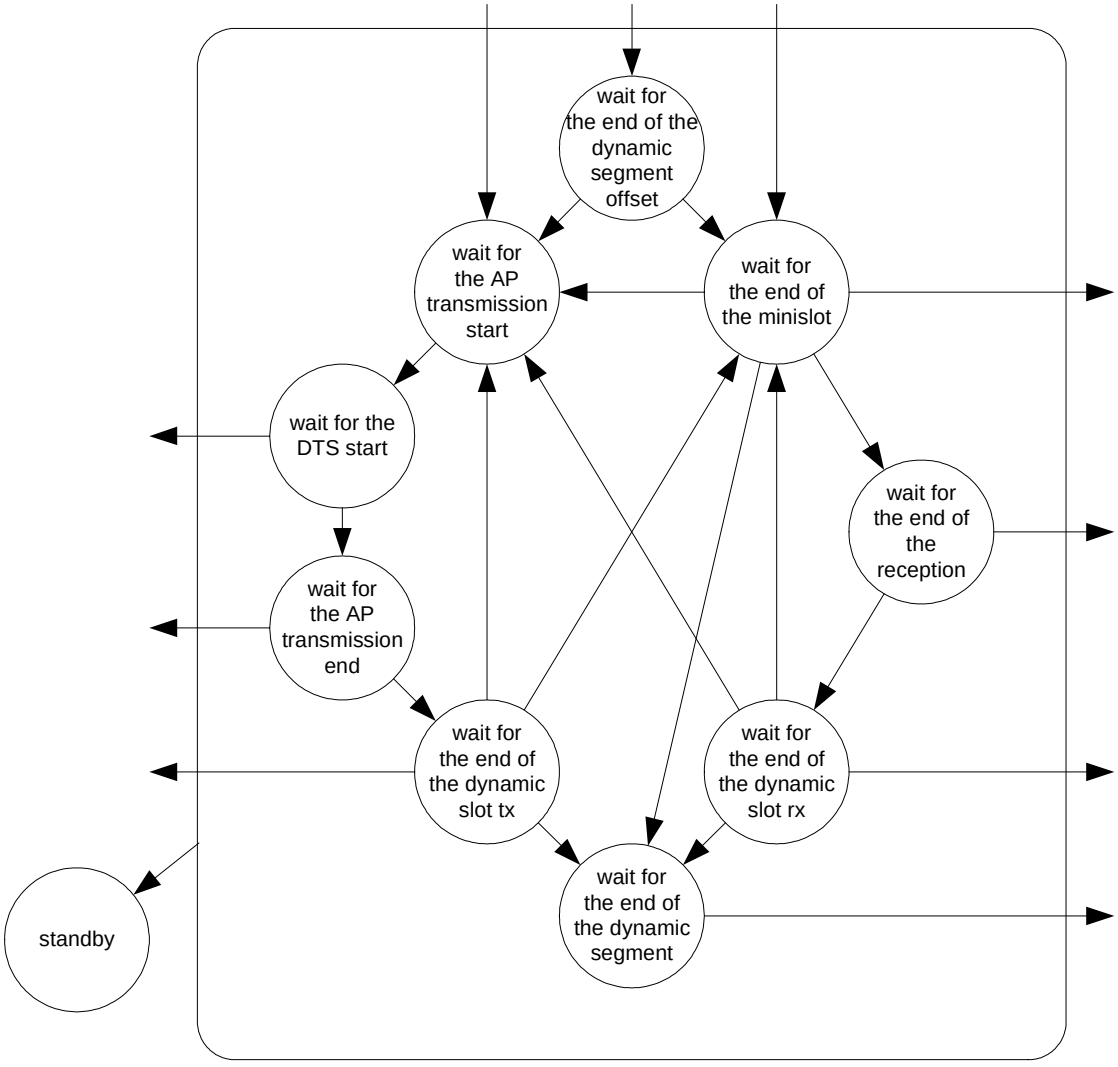

Figure 5-19: Media access in the dynamic segment [MAC_A].

The node shall perform media access in the dynamic segment as depicted in Figure 5-20, Figure 5-21, Figure 5-22, and Figure 5-23.

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.

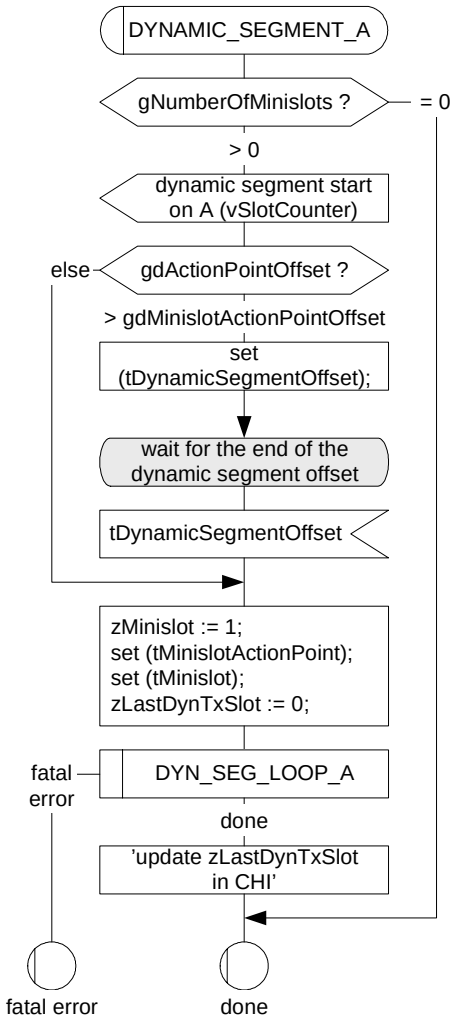

Figure 5-20: Media access in the dynamic segment start [MAC_A].

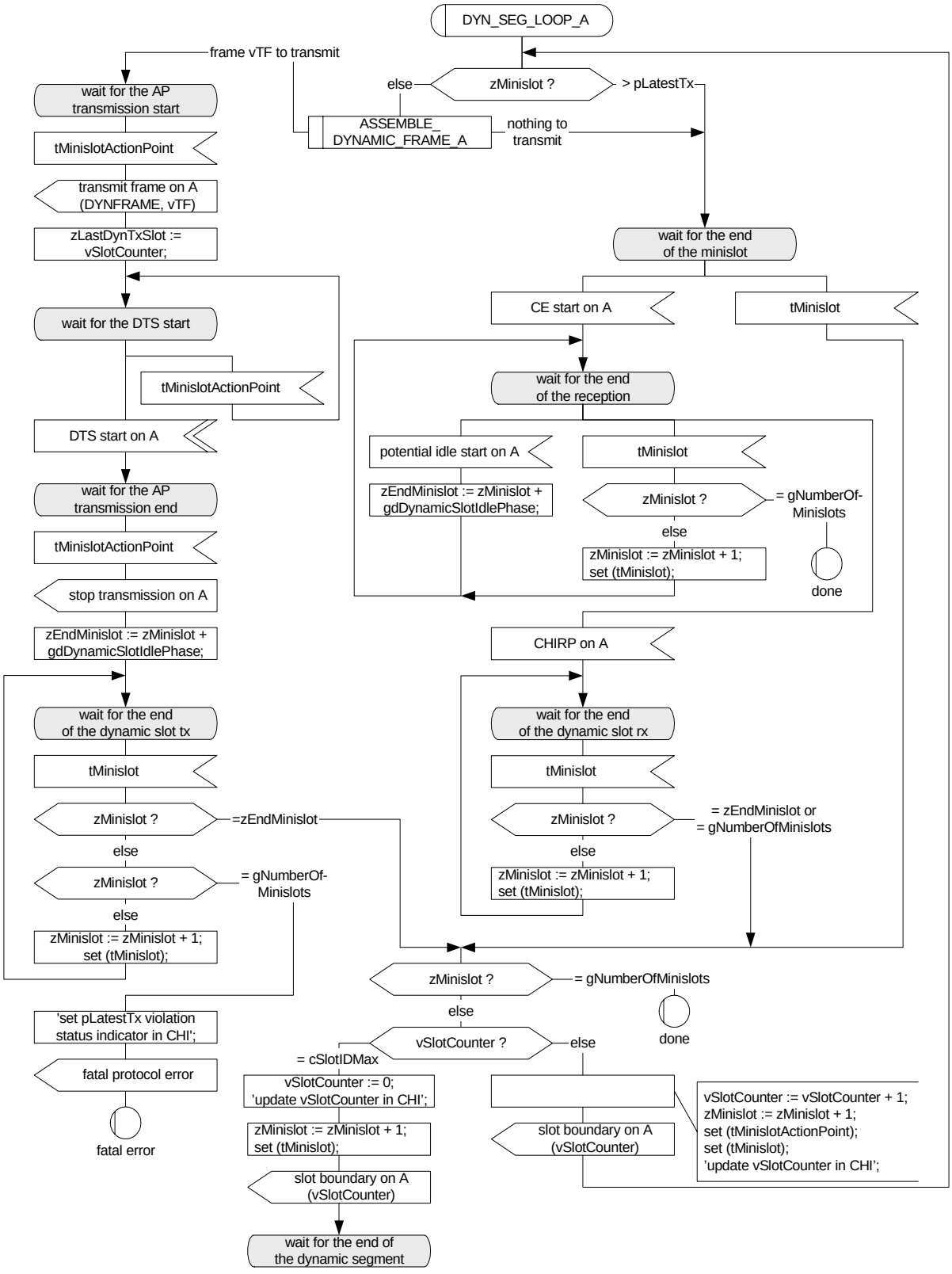

Figure 5-21: Media access in the dynamic segment arbitration [MAC_A][60].

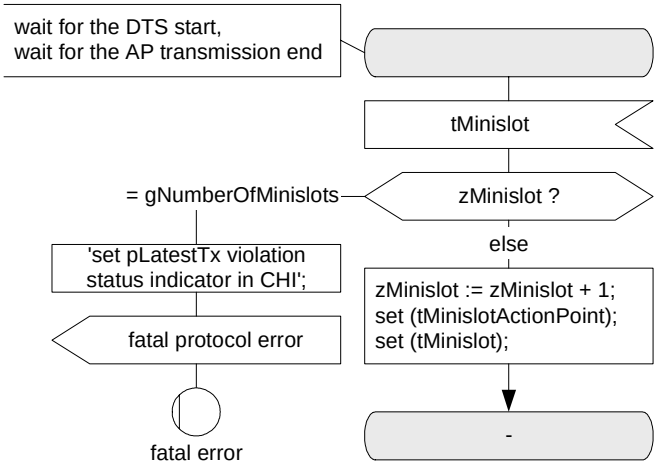

Figure 5-22: Minislot counting during transmission [MAC_A].

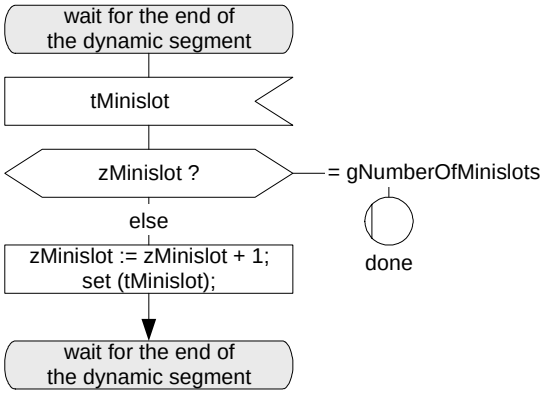

Figure 5-23: Dynamic segment termination in the event of slot counter exhaustion [MAC_A].

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_A. 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 the media access is operating in the ALL mode, if the dynamic segment has not exceeded the *pLatestTx* minislot (a node-specific upper bound), if the slot is assigned to the node, and if consistent payload data can be imported from the CHI.

Assuming a variable *vTCHI* of type *T_CHITransmission* 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.

[60] The implicit input *tMinislotActionPoint* in state *MAC:wait for the DTS start* has been explicitly defined in this figure to express more clearly its relationship to the priority input *DTS start on A*. In case both signals are received simultaneously the timer expiration is preserved for the subsequent state.

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*.

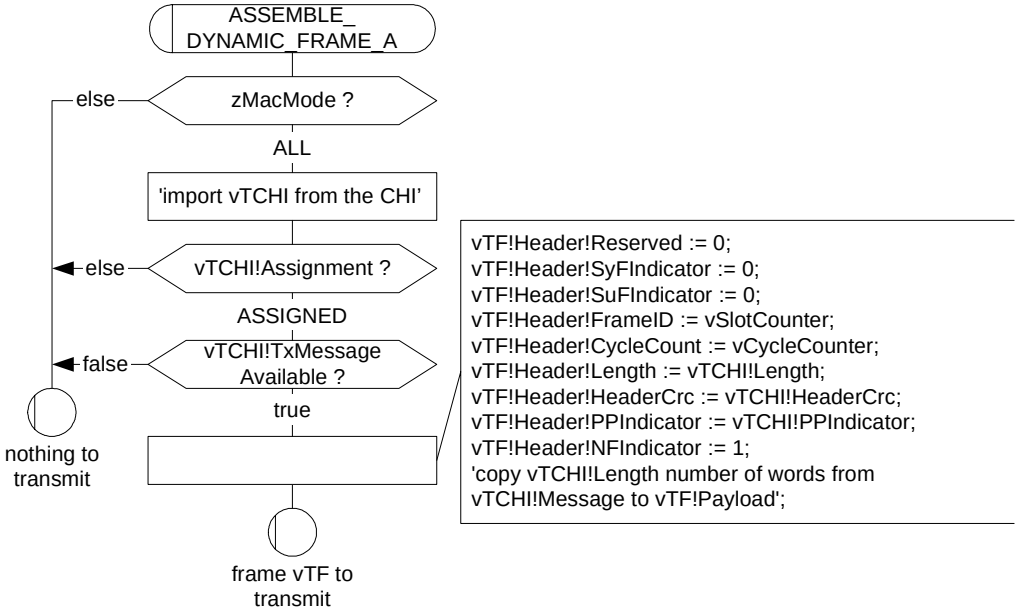

Figure 5-24: Frame assembly in the dynamic segment [MAC_A].

5.3.4 Symbol window related states

5.3.4.1 State machine for the symbol window media access control

Figure 5-25 gives an overview of the media access states in the symbol window and how they interrelate.

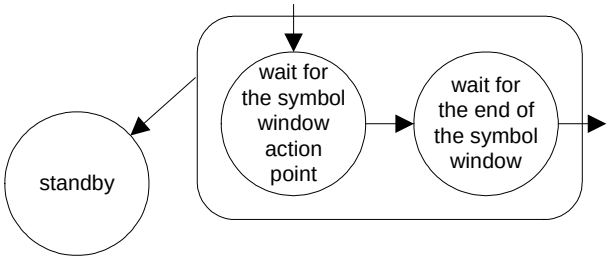

Figure 5-25: Media access in the symbol window [MAC_A].

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 symbol transmission conditions are met.

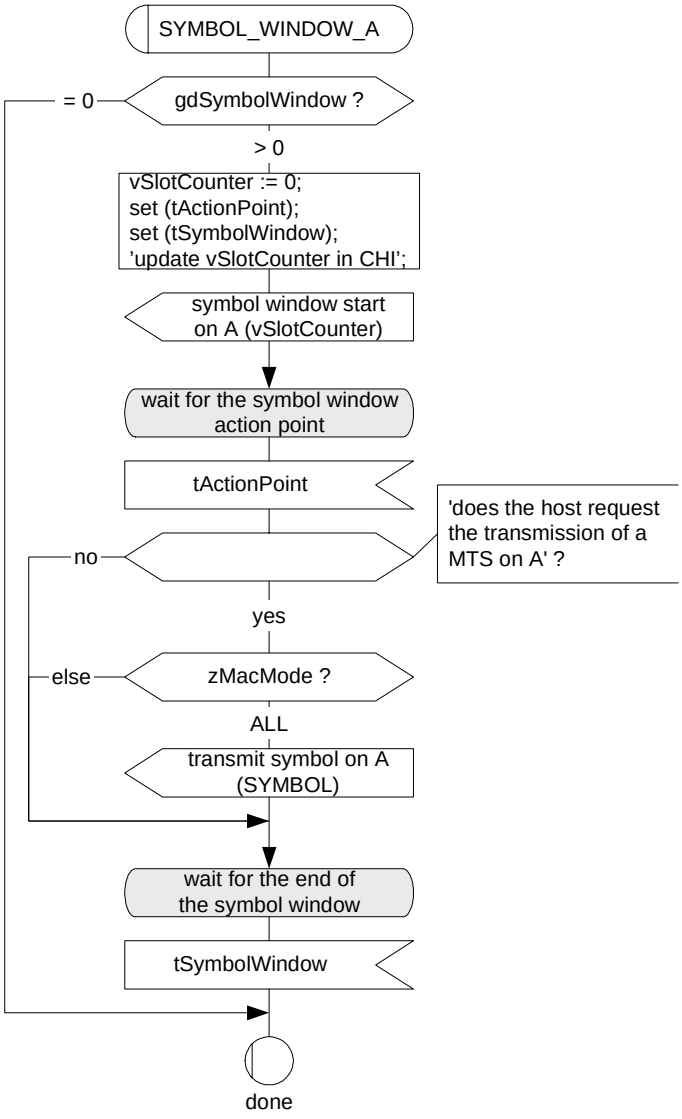

Figure 5-26: Media access in the symbol window [MAC_A].

5.3.4.2 Transmission condition in the symbol window

As in the two communication segments, whether or not a symbol shall be transmitted depends on the current protocol phase.

The node shall transmit a symbol on a channel if the media access is in the ALL mode and if a symbol is released for transmission.

5.3.5 Network idle time

Macro NIT_A in Figure 5-27 depicts the behavior at the start of the network idle time.

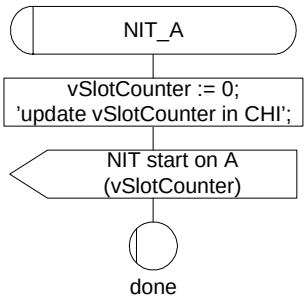

Figure 5-27: Network idle time [MAC_A].

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[61].

[61] 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.

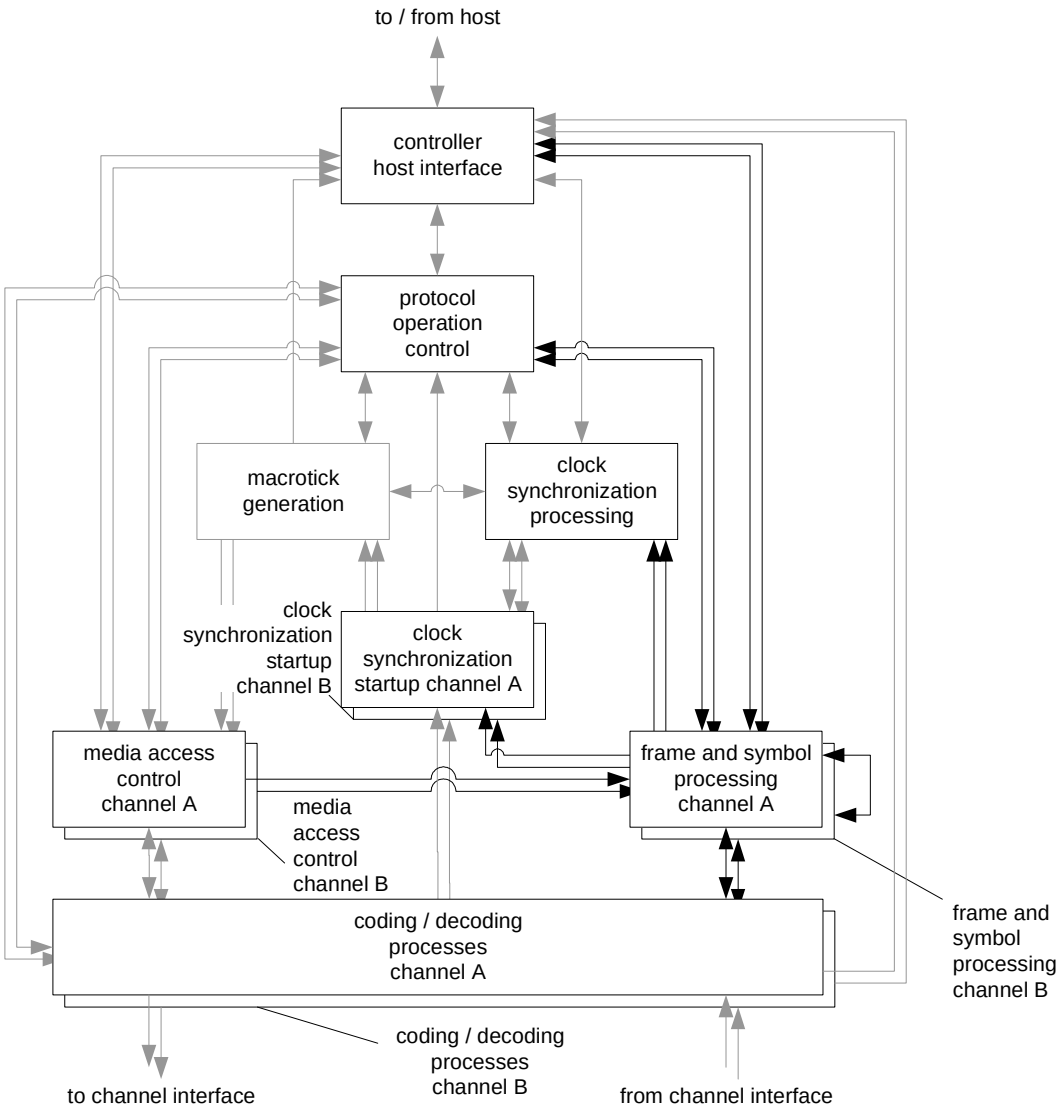

Figure 6-1: Frame and symbol processing context.

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.
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.

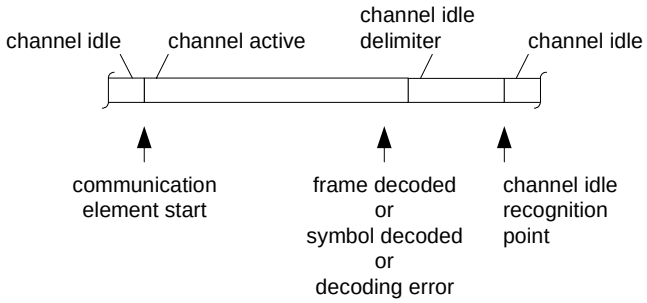

Figure 6-2: Reception-related events for FSP.

For communication channel A the reception relevant 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 6-2),
3. symbol decoded on channel A (signal *symbol decoded on A*),
4. decoding error on channel A (signal *decoding error on A*),
5. channel idle recognition point detected on channel A (signal *CHIRP on A*),
6. content error on channel B (signal *content error on B*)[62].

For communication channel B the reception relevant 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 6-2),
3. symbol decoded on channel B (signal *symbol decoded on B*),
4. decoding error on channel B (signal *decoding error on B*),
5. channel idle recognition point detected on channel B (signal *CHIRP on B*),
6. content error on channel A (signal *content error on A*).

Definition 6-2 gives the formal definition of the *T_ReceiveFrame* data structure.

```
newtype T_ReceiveFrame
      struct
```

[62] In order to address channel consistency checks for sync frames.

```
            PrimaryTRP  T_MicrotickTime;
            Channel     T_Channel;
            Header      T_Header;
            Payload     T_Payload;
endnewtype;
```

Definition 6-2: Formal definition of T_ReceiveFrame.

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.

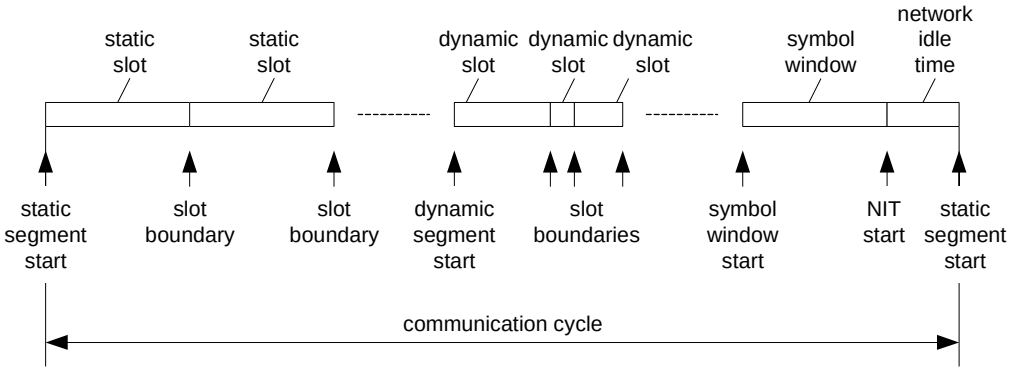

Figure 6-3: Timing-related events for FSP.

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.

The slot status consists of

1. a channel identifier that relates to the corresponding reception channel,
2. the value of the slot counter of the slot for the corresponding reception channel,
3. the value of the cycle counter of the slot,
4. an enumeration that denotes whether or not a valid communication element was received and its respective type,
5. a flag that denotes whether a syntax error was observed in the slot,
6. a flag that denotes whether a content error was observed in the slot,
7. a flag that denotes whether a boundary violation was observed in the slot,
8. a flag that denotes whether a transmission conflict was observed in the slot,
9. an enumeration that denotes to which segment the slot belongs.

Definition 6-3 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;
      TxConflict        Boolean;
      Segment           T_Segment;
endnewtype;
```

Definition 6-3: Formal definition of T_SlotStatus.

The parameter *Channel* holds the channel identifier.

The parameter *SlotCount* holds the value of the slot counter of the corresponding slot.

The parameter *CycleCount* holds the value of the cycle counter of the corresponding cycle.

The parameter *ValidFrame* denotes whether a valid frame was received in a slot of the static or dynamic segment. The parameter is set to false if no valid frame was received, or to true if a valid frame was received.

The parameter *ValidMTS* denotes whether a valid MTS was received in the symbol window. The parameter is set to false if no valid MTS was received, or to true if a valid MTS was received.

The parameter *SyntaxError* denotes whether a syntax error has occurred. A syntax error occurs if

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 symbol 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. two or more symbols are received within the symbol window.

This parameter 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.

The parameter *ContentError* denotes whether a content error has occurred. A content error occurs if

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 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,
4. 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,
5. in the dynamic segment the sync frame indicator contained in the header of the received frame is set to one,
6. in the dynamic segment the startup frame indicator contained in the header of the received frame is set to one,
7. in the dynamic segment the null frame indicator contained in the header of the received frame is set to zero.

This parameter 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.

The parameter *BViolation* denotes whether a boundary violation occurred at either boundary 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 parameter is set to false if no boundary violation occurred, or to true if a boundary violation did occur at either the beginning or end of the slot.

See Table 9-2 for a description of the meaning of various combinations of *ValidFrame*, *SyntaxError*, *ContentError*, and *BViolation*.

The parameter *TxConflict* denotes whether reception was ongoing at the time the node started a transmission. The parameter is set to false if reception was not ongoing, or to true if reception was ongoing.

The parameter *Segment* denotes the segment in which the slot status was recorded. Its type is defined in Definition 6-4.

```
newtype T_Segment
      literals STUP, STATIC, DYNAMIC, SW, NIT;
endnewtype;
```

Definition 6-4: Formal definition of T_Segment.

The parameter *Segment* is set to STUP during the unsynchronized 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.

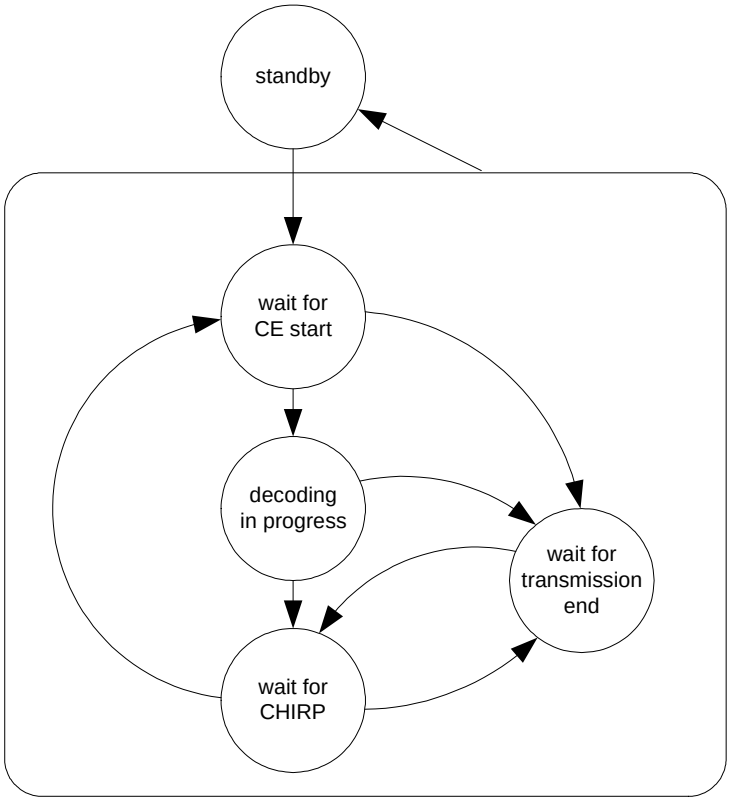

Figure 6-4: State overview of the FSP state machine (shown for one channel).

6.3.1 Initialization and state *FSP:standby*

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.

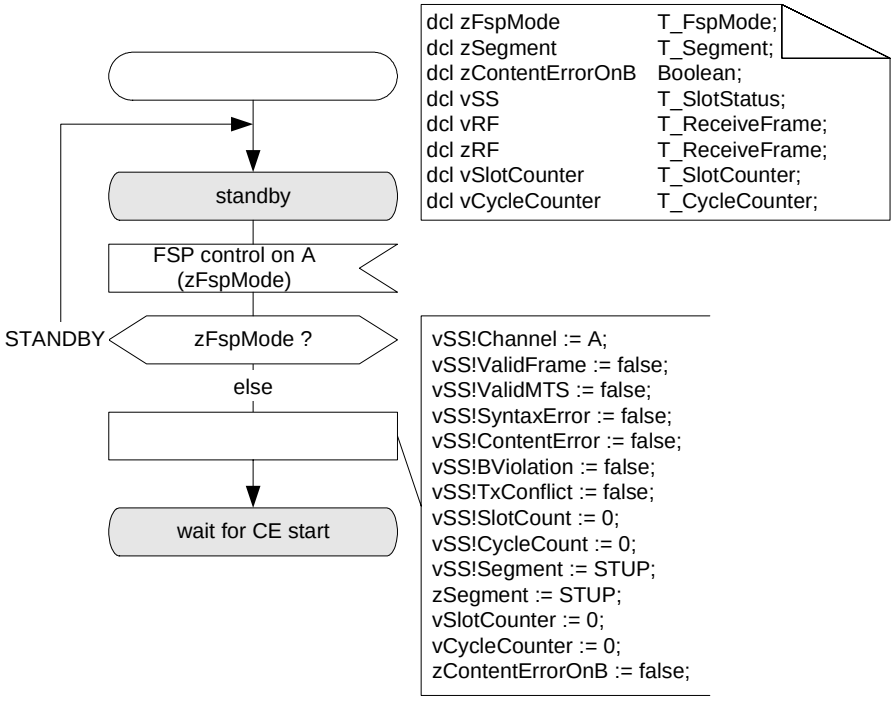

Figure 6-5: FSP process [FSP_A].

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.

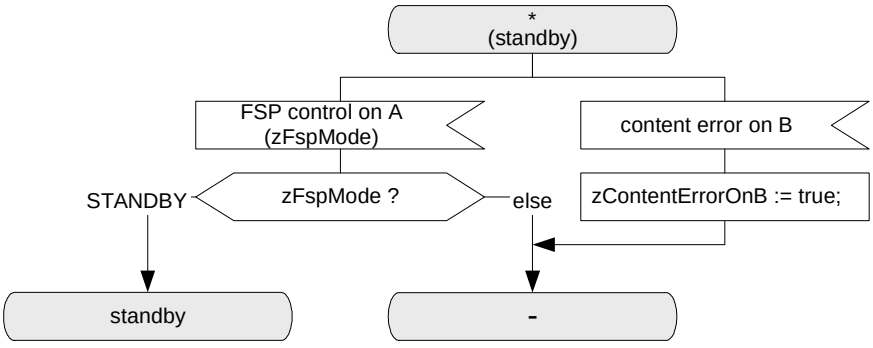

Figure 6-6: FSP control [FSP_A].

In addition, the node shall apply cross-channel content checks to identify cross-channel inconsistencies whenever *zSegment* = STATIC.[63]

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.

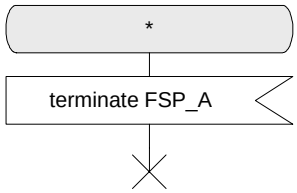

Figure 6-7: Termination of the FSP process [FSP_A].

6.3.2 Macro SLOT_SEGMENT_END_A

The macro SLOT_SEGMENT_END_A that is depicted in Figure 6-8 shall be called within the FSP process

1. at the end of each static slot,
2. at the end of each dynamic slot, if dynamic slots are configured,
3. at the end of the symbol window, if the symbol window is configured, 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.

A node shall initialize the slot status *vSS* for aggregation in the subsequent slot.

[63] *zSegment* is STATIC during the static segment in normal operation, but can also be STATIC during some portions of startup.

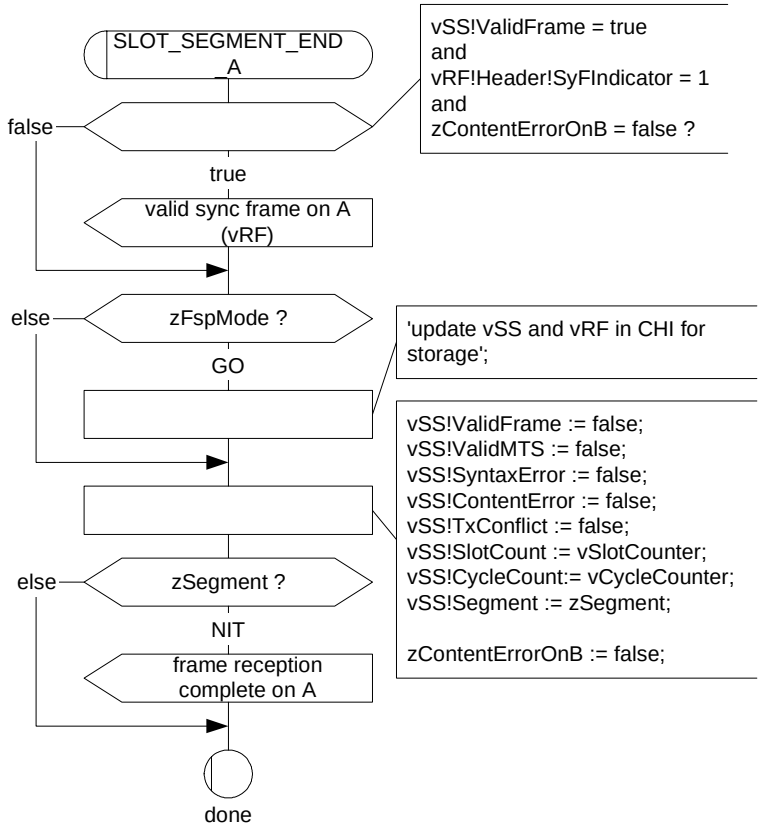

Figure 6-8: Slot and segment end macro [FSP_A].

6.3.3 State *FSP:wait for CE start*

The *FSP:wait for CE start* state and the transitions out of this state are depicted in Figure 6-9.

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_A 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.

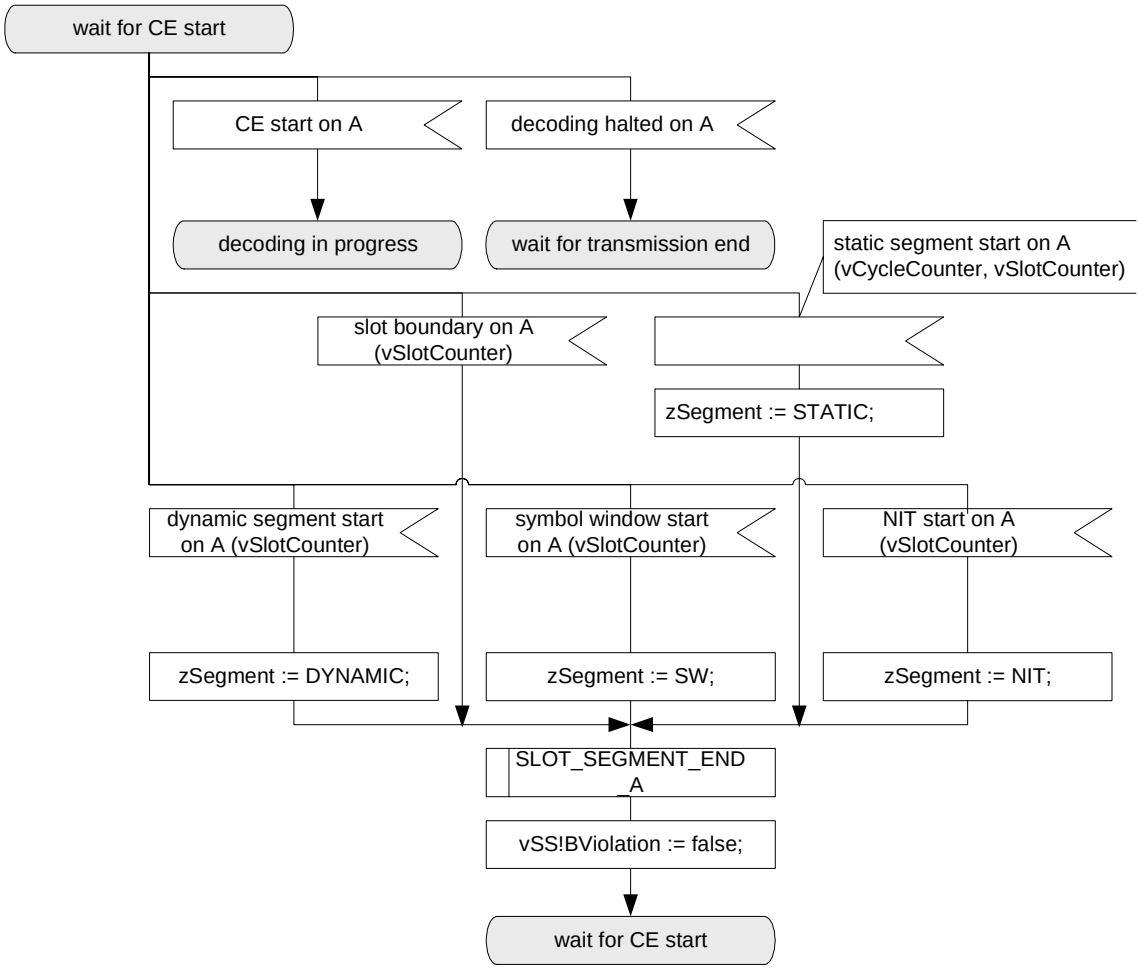

Figure 6-9: Transitions from the *FSP:wait for CE start* state [FSP_A].

6.3.4 State *FSP:decoding in progress*

The *FSP:decoding in progress* state and the transitions out of this state are depicted in Figure 6-10 and Figure 6-11.

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 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_A macro to provide the current slot status, and any frame data that may have been received, to the host interface for further processing.

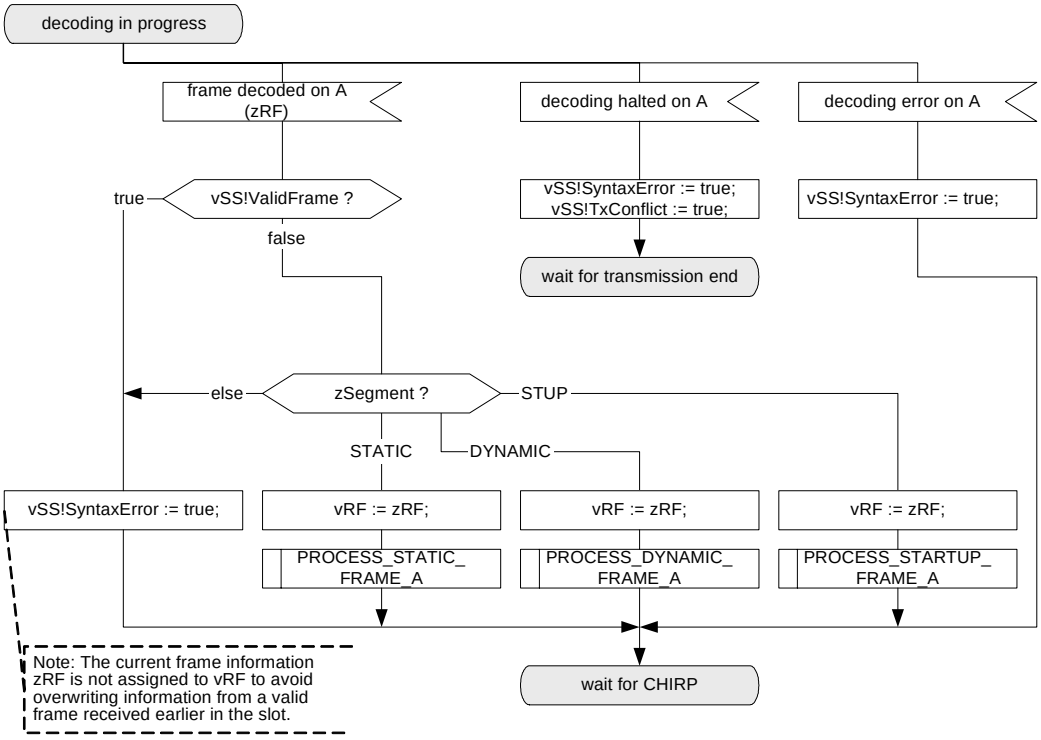

Figure 6-10: Transitions from the *FSP:decoding in progress* state [FSP_A].

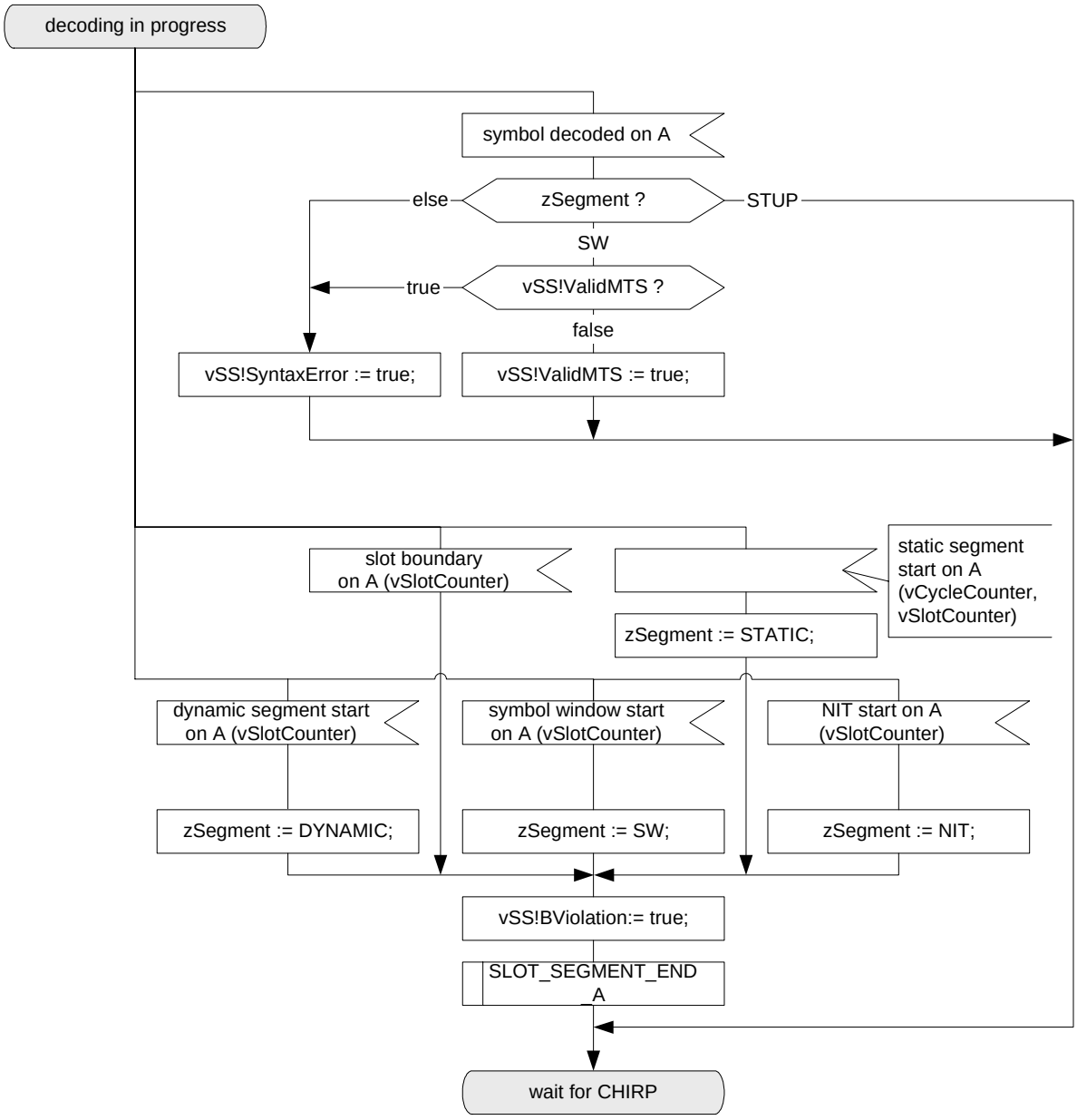

Figure 6-11: Transitions from the *FSP:decoding in progress* state [FSP_A].

6.3.4.1 Frame reception checks during non-TDMA operation

The frame acceptance checks that the node shall apply during non-TDMA operation are defined in the macro PROCESS_STARTUP_FRAME_A depicted in Figure 6-12.

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 sync frame indicator included in the header is set to one.

3. The startup frame indicator included in the header is set to one.
4. The payload length included in the header of the frame equals the globally configured length for static frames *gPayloadLengthStatic*.

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*.

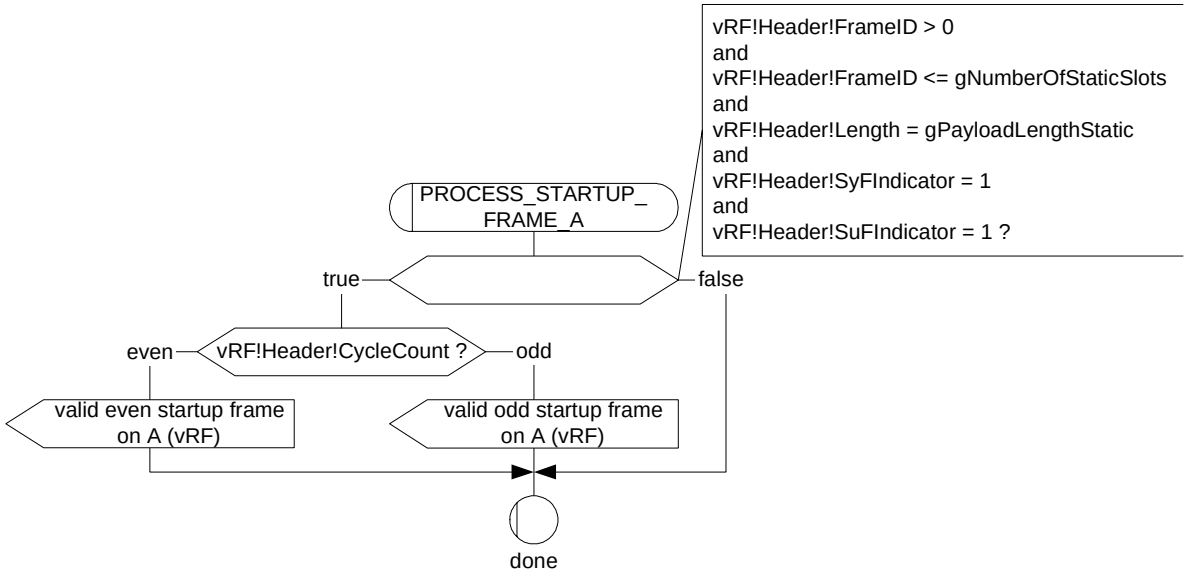

Figure 6-12: Frame acceptance checks during non-TDMA operation [FSP_A].

6.3.4.2 Frame reception checks during TDMA operation

6.3.4.2.1 Frame reception checks in the static segment

Figure 6-13 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 TDMA operation in the static segment are defined in the macro PROCESS_STATIC_FRAME_A depicted in Figure 6-14.

For each configured communication channel the node shall accept the first frame that fulfills 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*.

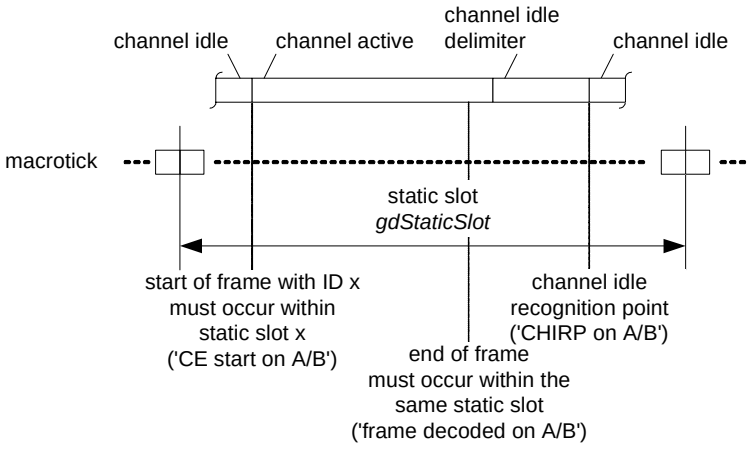

Figure 6-13: Frame reception timing for a static slot.

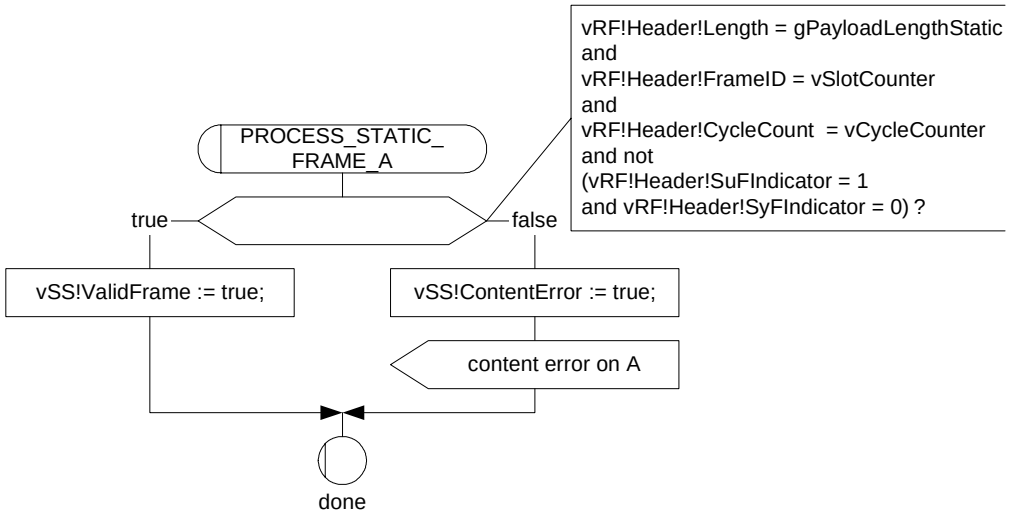

Figure 6-14: Frame acceptance checks for the static segment [FSP_A].

6.3.4.2.2 Frame reception checks in the dynamic segment

Figure 6-15 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 TDMA operation in the dynamic segment are defined in the macro PROCESS_DYNAMIC_FRAME_A depicted in Figure 6-16.

For each configured communication channel the node shall accept the first frame that fulfills 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.

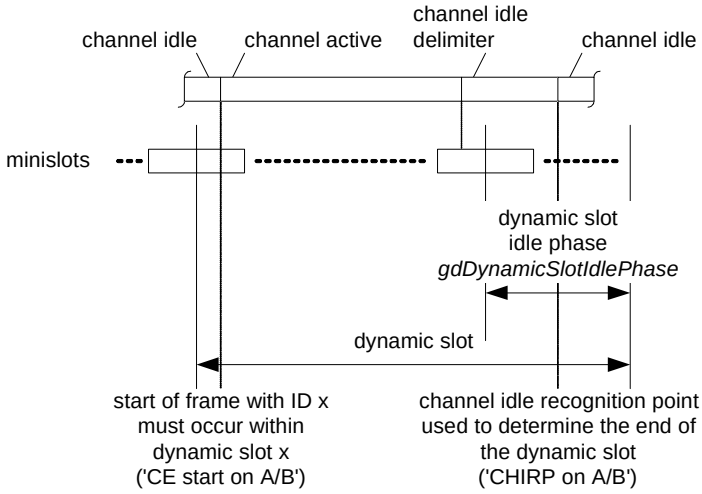

Figure 6-15: Frame reception timing for a dynamic slot.

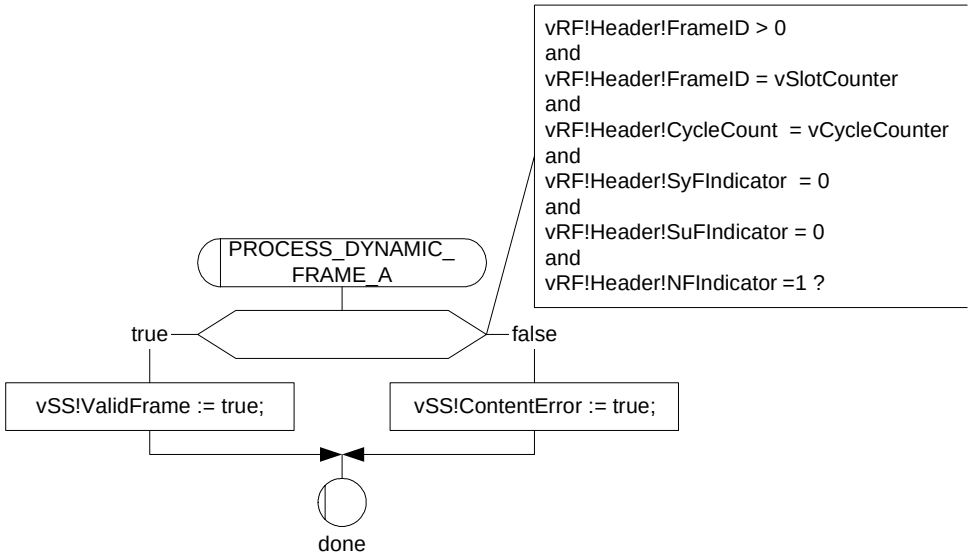

Figure 6-16: Frame acceptance checks for the dynamic segment [FSP_A].

6.3.5 State *FSP:wait for CHIRP*

The *FSP:wait for CHIRP* state and the transitions out of this state are depicted in Figure 6-17.

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_A 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.

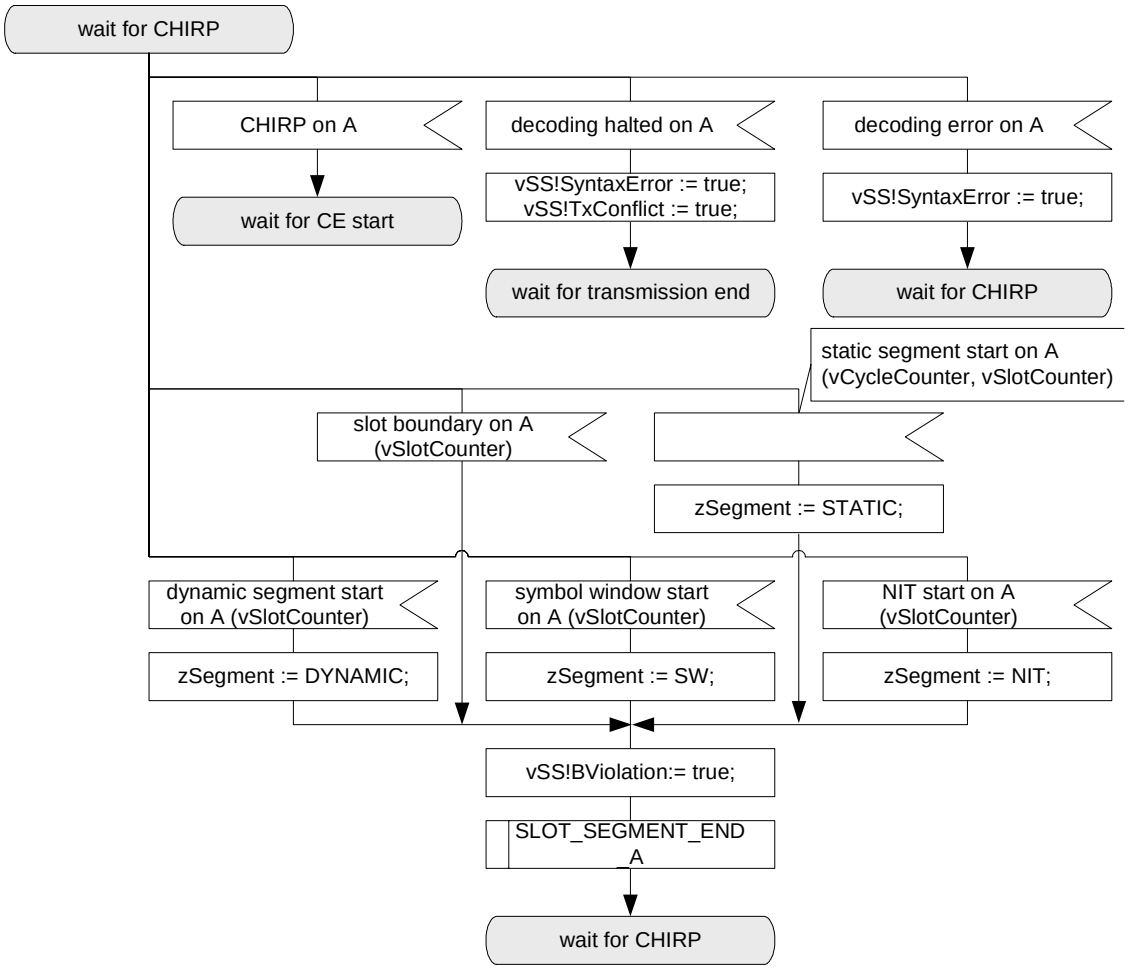

Figure 6-17: Transitions from the *FSP:wait for CHIRP* state [FSP_A].

6.3.6 State *FSP:wait for transmission end*

The *FSP:wait for transmission end* 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 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 error to the protocol operation control process.

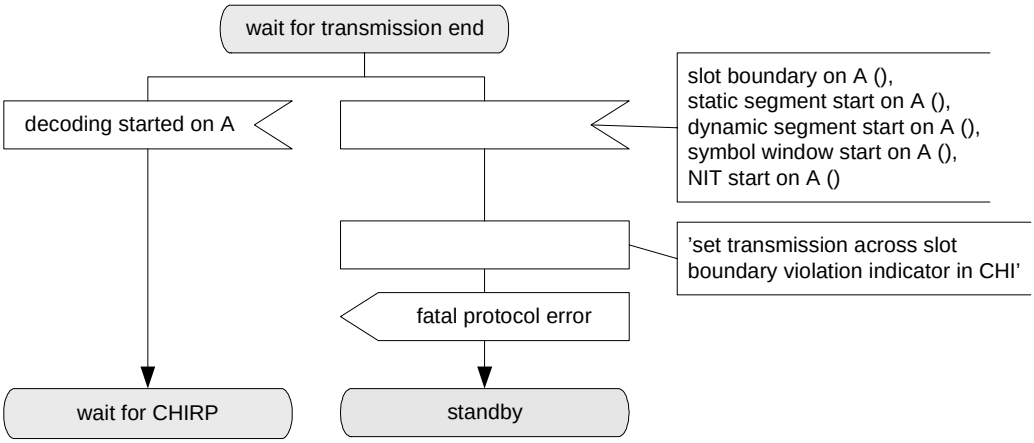

Figure 6-18: Transitions from the *FSP:wait for transmission end* state [FSP_A].

Chapter 7
Wakeup and Startup

This chapter describes the procession of a FlexRay cluster from sleep mode to full operation and the integration of newly configured nodes into a FlexRay cluster already in operation.

First the cluster wakeup is described in detail. Some application notes regarding the interaction between communication controller and host are provided.

Following the wakeup section, communication startup and reintegration is described. This section also describes the integration of nodes into a communication cluster.

7.1 Cluster wakeup

This section describes the procedure[64] used by communication controllers to initiate the cluster *wakeup*.

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[65]. 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[66] since these nodes cannot give feedback until the startup phase. The host shall be aware of possible failures of the wakeup and act accordingly.

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 section 7.1.4.1).

[64] 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.

[65] 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.

[66] For example, the transmission unit of the bus driver may be faulty.

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[67].

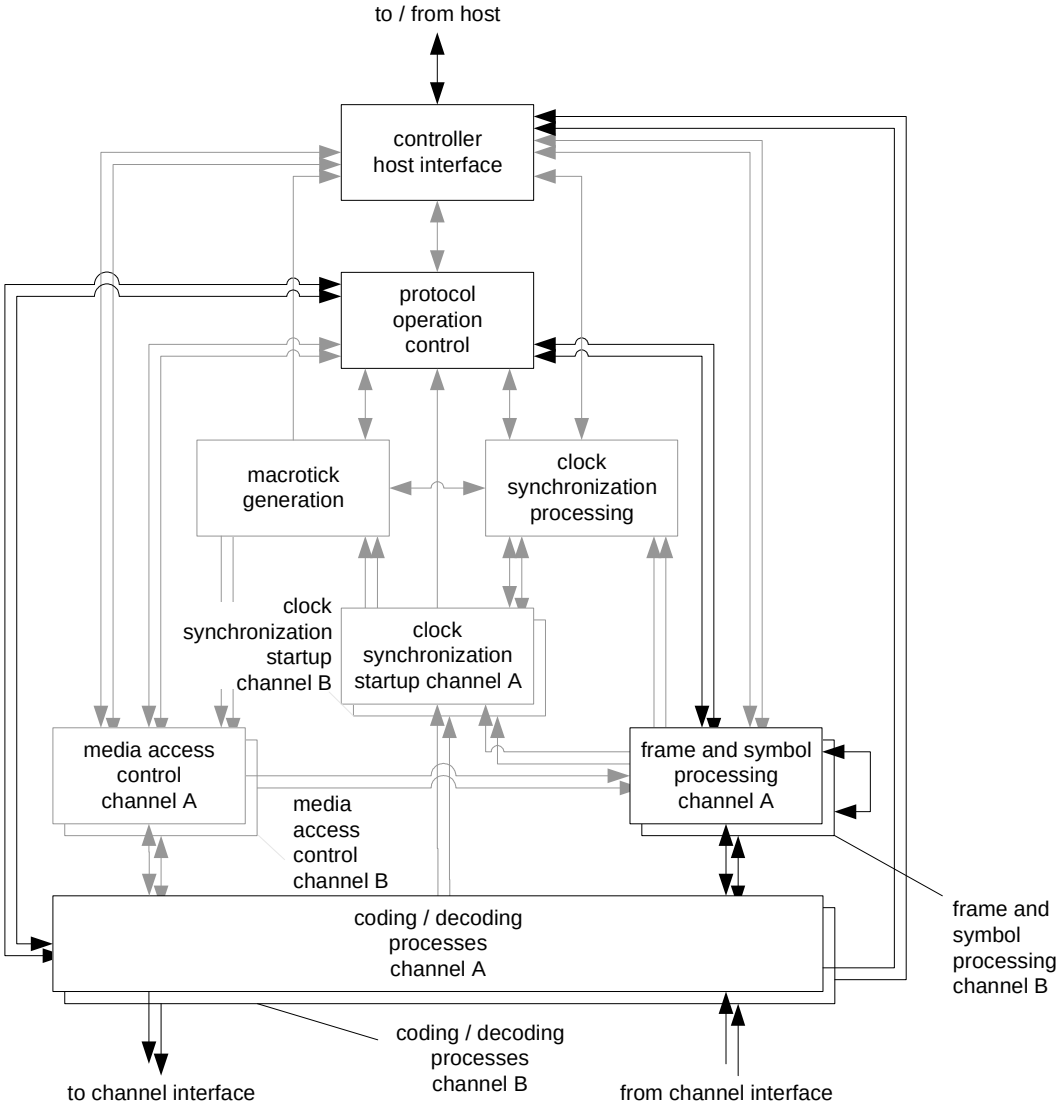

Figure 7-1: Protocol operation control context.

[67] 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.

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

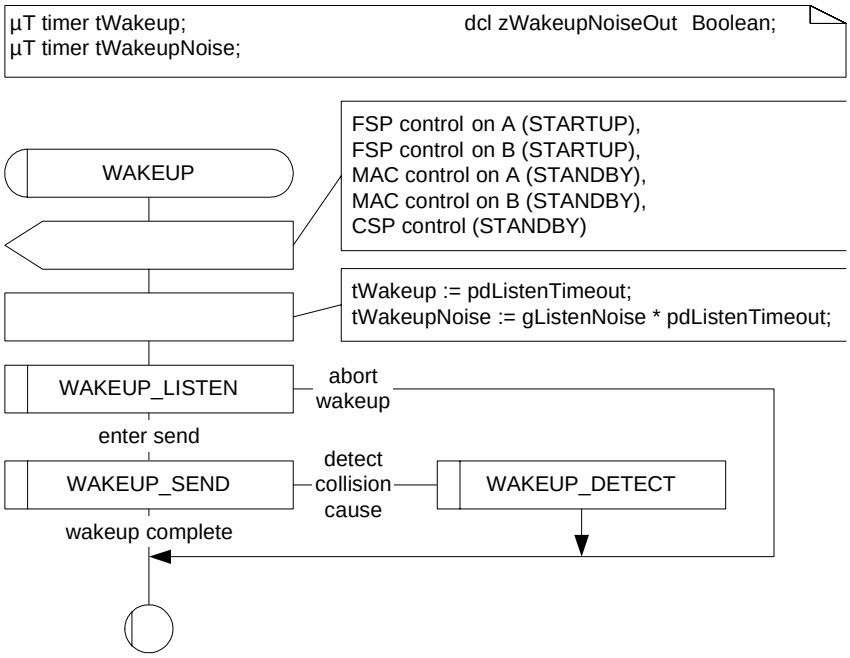

Figure 7-2: Structure of the wakeup state machine [POC].

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 communication controller has not yet executed the WAKEUP mechanism since the last entry to the *POC:default config* state (see Figure 2-7), or when the POC responds to a CHI WAKEUP command,

- 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.

7.1.3.2 The *POC:wakeup listen* state

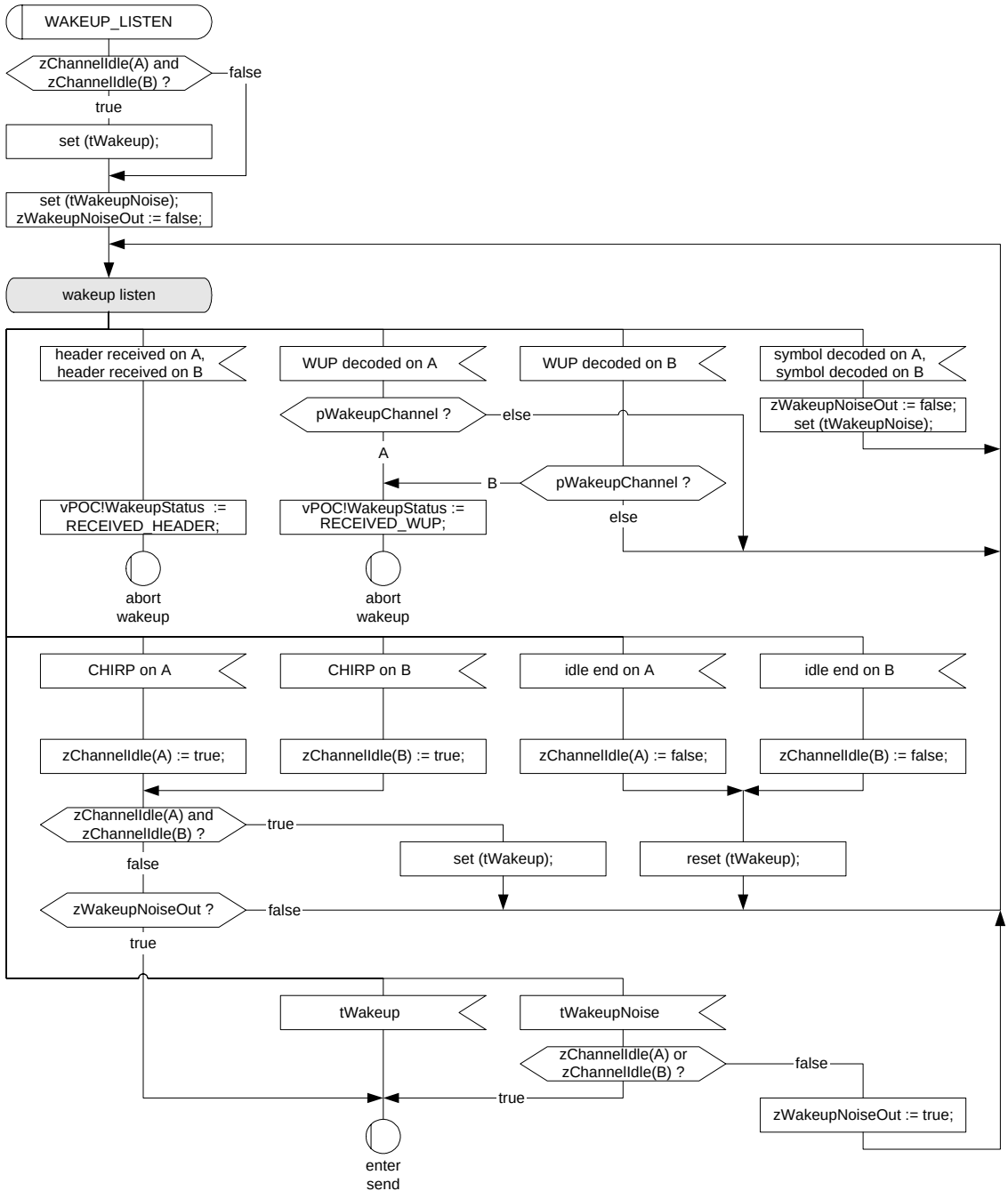

Figure 7-3: Transitions from the *POC:wakeup listen* state [POC][68].

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.

[68] For a single channel node, the unattached channel is assumed to always be idle.

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.

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

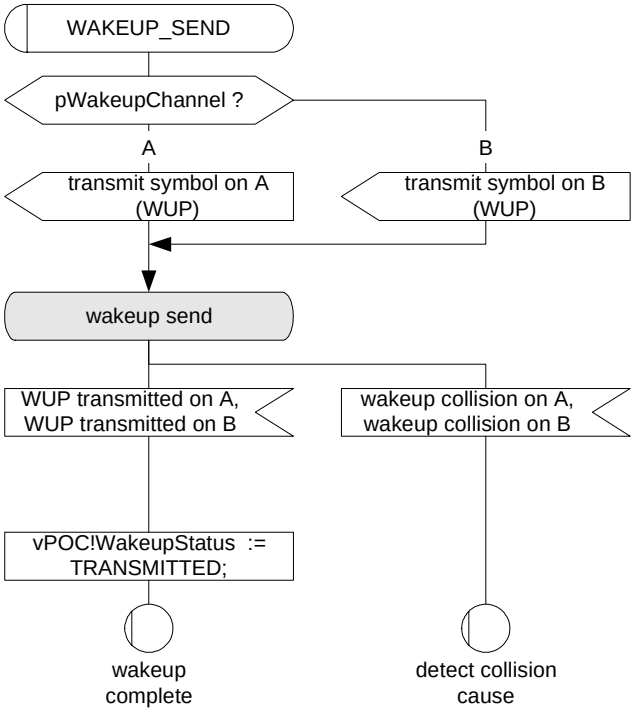

Figure 7-4: Transitions from the state *POC:wakeup send* state [POC].

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

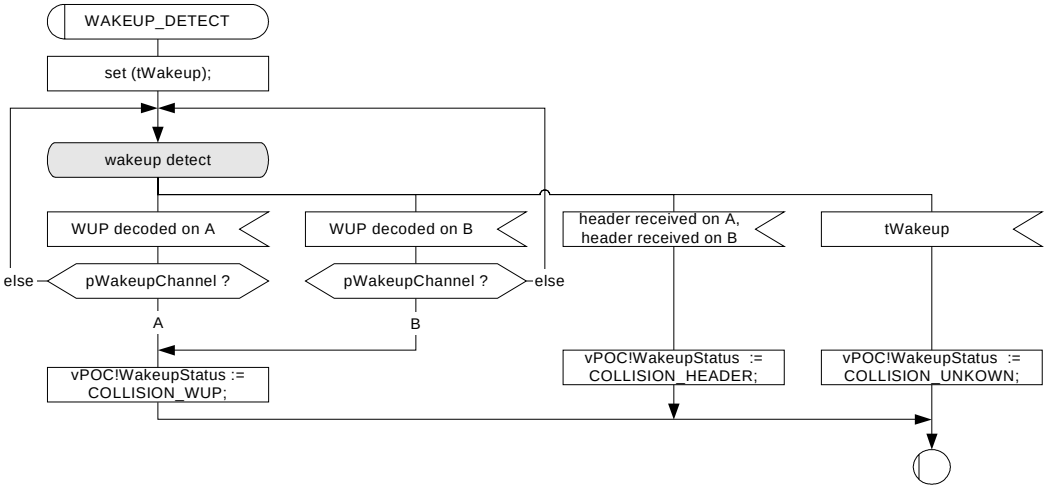

Figure 7-5: Transitions from the state *POC:wakeup detect* state [POC].

In this state, the communication controller attempts to discover the reason for the wakeup collision encountered in the previous state (*POC:wakeup send*).

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.

7.1.4 Wakeup application notes

The host must coordinate the bus driver and the communication controller wakeup modes. It must coordinate the wakeup of the two channels and must decide whether or not to wake a specific channel. Since proper behavior of the host is important to the wakeup procedure, the required host behavior is described here.

7.1.4.1 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[69].

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.[70]

A single-channel node in a dual-channel cluster can trigger a cluster wake-up 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.

[69] 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.

[70] 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.

7.1.4.1.1 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. The coldstart inhibit mode[71] shall be entered by setting *vColdStartInhibit* to true.
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 9) instead of performing a wakeup.
   If no wakeup pattern 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.
8. The host commands the communication controller to leave the coldstart inhibit mode and to commence startup.

7.1.4.1.2 Dual-channel nodes

This section describes the wakeup behavior of dual-channel nodes in dual-channel clusters.

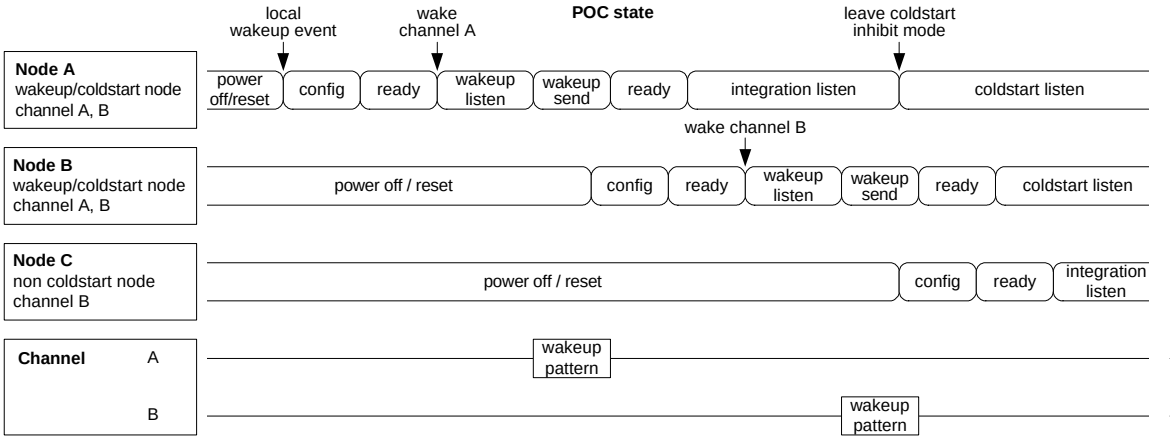

Figure 7-6: A short example of how the wakeup of two channels can be accomplished in a fault-tolerant way by coldstart nodes.[72]

A communication controller shall not send a wakeup pattern on both channels at the same time[73]. If it is necessary to wake both channels, the host can only wake them one at a time.

[71] This has no functional effect for non-coldstart nodes.
[72] 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.
[73] This requirement ensures that an erroneous communication controller cannot disturb all ongoing communication by transmitting wakeup patterns on both channels at the same 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 7-6). Following this, it does not wake the other channel but rather enters startup. If it is a coldstart node, the coldstart inhibit flag must be set to true, so that it cannot actively initiate the cluster startup. Only after the node detects a wakeup pattern on channel B should it clear the coldstart inhibit flag and actively coldstart the cluster.

Two example wakeup strategies are now given as examples to demonstrate how cluster wakeup can be accomplished. Neither strategy is concerned with the details of error recovery and therefore do not handle some error situations that are likely to occur.

7.1.4.1.2.1 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. The coldstart inhibit mode[74] shall be set.
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).
4. If both channels are awake, the host can proceed to startup (step 11).
   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 activates the bus driver of *pWakeupChannel*.
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 11).
   If the result of the wakeup attempt is RECEIVED_HEADER or COLLISION_HEADER, the host can assume that both channels are awake. It activates any remaining sleeping bus driver and proceeds to startup (step 11).
   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_UNKOWN, 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 command the communication controller to leave the coldstart inhibit mode. Otherwise, it waits 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 *pWakeupChannel* cannot be listened to during the *POC:wakeup listen* state. If the bus driver of the channel is subject to an incoming link failure, ongoing communication might be disturbed and the node would not come up without additional error recovery strategies.

[74] This has no functional effect for non-coldstart nodes.

7.1.4.1.2.2 Wakeup pattern reception by the communication controller

The wakeup pattern receiver of the communication controller is active as long as the CODEC is in WAKEUP mode. During this time the reception of a wakeup pattern will be directly signaled to the CHI and from there to the host.

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. The coldstart inhibit mode[75] shall be set.
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_UNKOWN, 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 command the communication controller to leave the coldstart inhibit mode. Otherwise, it waits until the wakeup pattern receiver 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). If the communication controller internal wakeup pattern detector receives a wakeup pattern on this channel, it is further 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.

7.1.4.2 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.

7.1.4.2.1 Frame header reception without coding violation

When a frame header without coding violation 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.

[75] This has no functional effect for non-coldstart nodes.

7.1.4.2.2 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.

7.1.4.2.3 Wakeup pattern transmission

The communication controller has transmitted the complete wakeup pattern on channel *pWakeupChannel*. The node can now proceed to startup. In a dual-channel cluster, coldstart nodes may need to be put into the coldstart inhibit mode (see section 7.1.4.1).

7.1.4.2.4 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 EMC disturbances on the bus or an internal error.[76]

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 7.1.4.3.

7.1.4.3 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[77]). 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.

7.1.4.4 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 initiating the wakeup by the minimal amount of time it takes another coldstart node to become awake, to be configured, and to enter startup[78]. 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 7.1.4.3).

[76] 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.

[77] E.g. an erroneous star that needs significantly more time to start up and to be able to forward messages.

[78] This parameter depends heavily on implementation details of the components used and the ECU structure.

The *vColdstartInhibit* flag can be used to effectively deal with this situation. A communication controller with this flag set to true will only participate in the startup attempts made by other nodes but not initiate one itself. The host should not clear this flag before the above mentioned minimal time for another node to become ready for the communication startup. However, the host shall clear it as soon as all coldstart nodes are awake in the fault-free case[79]. Please refer to section 7.2.2 for further details of this flag.

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*.

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 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). In each cluster consisting of at least three nodes, at least three nodes shall be configured to be coldstart nodes[80]. In clusters consisting of less than three 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. The parameter *gColdStartAttempts* shall be configured to be at least two. At least two fault-free coldstart nodes are necessary for the cluster to start up.

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*.

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 Chapter 2). 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 7.1.4).

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.
- Each of the coldstart nodes finishes its startup as soon as stable communication with one of the other coldstart nodes is established.

[79] If these times are not known at system design time, it is advised to clear the flag late rather than early.
[80] This does not imply any restrictions concerning which nodes may initiate a cluster wakeup.

7.2.1.2.2 Integration of the non-coldstart nodes

- A non-coldstart node requires at least two startup frames from distinct nodes for integration. This condition ensures that each non-coldstart node always joins the majority of the coldstart nodes[81].
- Integrating non-coldstart nodes may start their integration before coldstart nodes have finished their startup.
- Integrating non-coldstart nodes may not finish their startup until at least two coldstart nodes have finished their startup.

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 7-7.

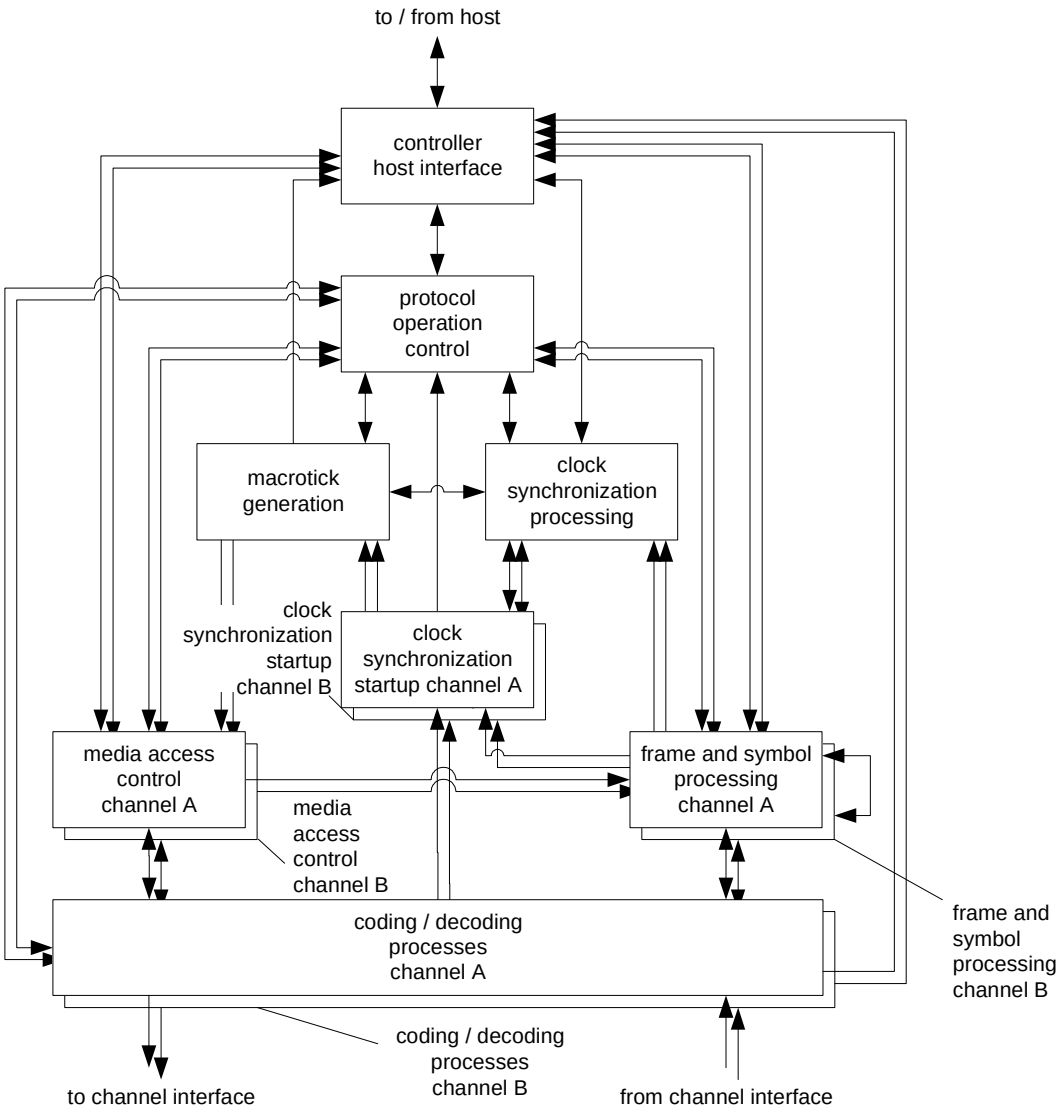

Figure 7-7: Protocol operation control context.

[81] This is the case under the assumption that the cluster contains exactly the three recommended coldstart nodes.

7.2.3 Coldstart inhibit mode

A FlexRay node shall support a coldstart inhibit mode. In *coldstart inhibit mode* the node shall be prevented from initializing the TDMA communication schedule. The communication controller shall start in the coldstart inhibit mode after leaving the *POC:config* state. Until the host gives the command to leave the coldstart inhibit mode, the communication controller shall not be allowed to assume the role of leading coldstart node, i.e., entering the coldstart path is prohibited. The node is still allowed to integrate into a running cluster or to transmit startup frames after another coldstart node has started the initialization of cluster communication.

Once the node is synchronized and integrated into cluster communication, *vColdstartInhibit* does not restrict the node's ability to transmit frames.

This mode can be used either to completely prohibit active startup attempts of a node, or to only delay them. It may be used 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.

7.2.4 Startup state diagram

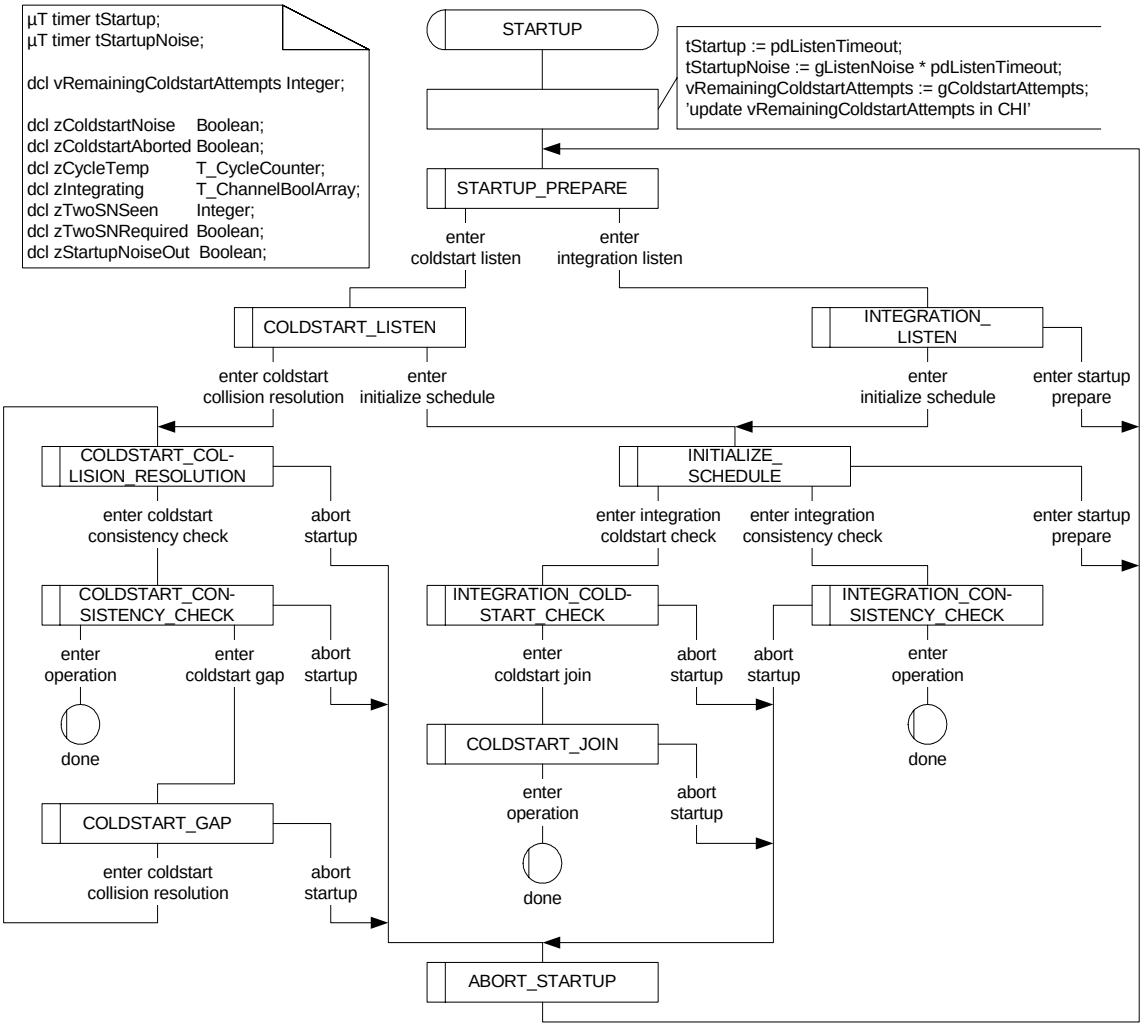

Figure 7-8: Startup state diagram [POC].

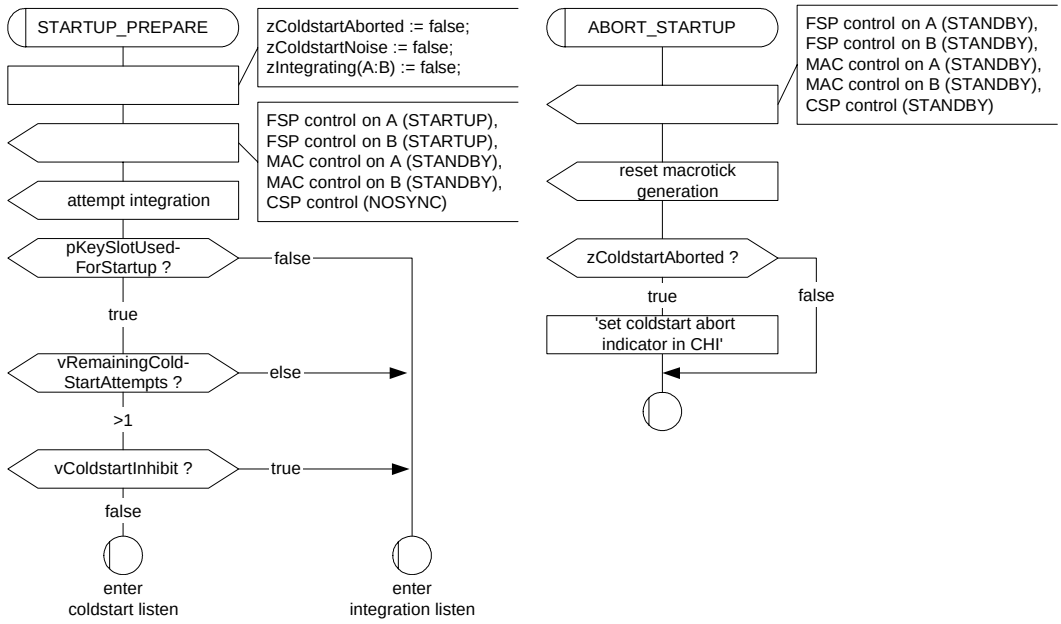

Figure 7-9: Helpful macros for startup [POC].

A communication controller can enter communication via three different paths. Two of these paths are restricted to coldstart nodes, while the remaining path is open only to non-coldstart nodes.

In sections 7.2.4.1, 7.2.4.2 and 7.2.4.3, an outline of the three startup paths in the fault-free case is given to supplement the precise SDL description that is given in the subsequent sections.

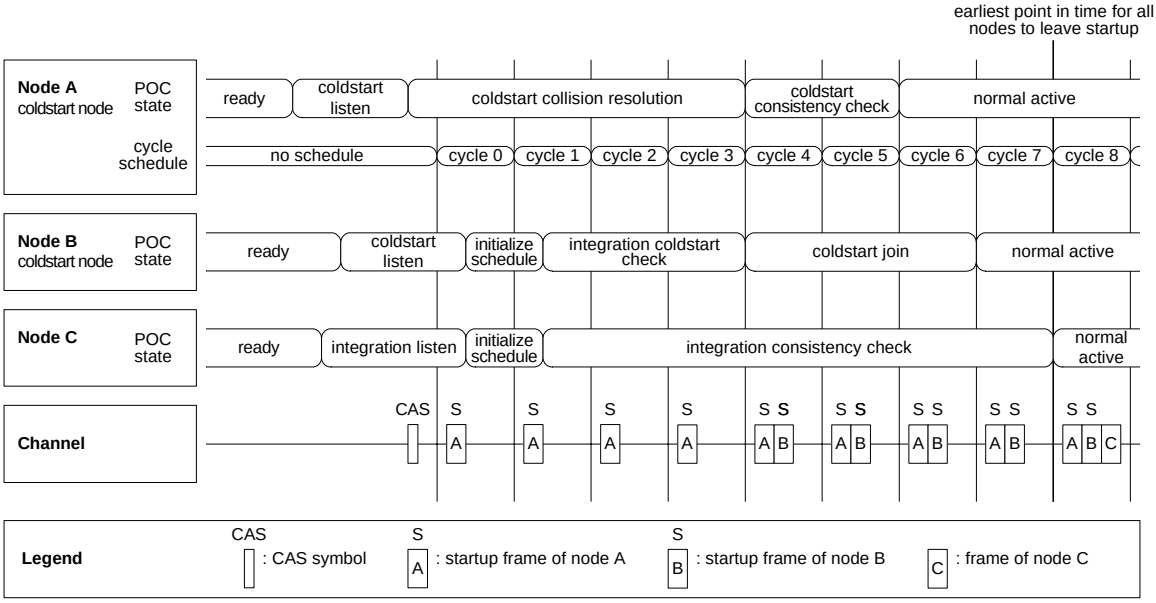

Figure 7-10: Example of state transitions for a fault-free startup[82].

[82] 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).

7.2.4.1 Path of the node initiating the coldstart (leading coldstart node)

Node A in Figure 7-10 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 7-11).

If no communication[83] 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 7-12).

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 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 7-13).

7.2.4.2 Path of the integrating coldstart nodes (following coldstart nodes)

Node B in Figure 7-10 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 7-11).

If communication[84] is received, it tries to integrate to a transmitting coldstart node[85]. 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 7-15).

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 7-16).

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 7-17).

7.2.4.3 Path of a non-coldstart node

Node C in Figure 7-10 follows 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 7-18).

If communication[86] 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 7-15).

[83] See Figure 7-11 for exact definition.
[84] See Chapter 8 for exact definition.
[85] Presumably it is the node that initiated the coldstart, but not necessarily.
[86] See Chapter 8 for exact definition.

In the following double cycles, it tries to find at least two coldstart nodes that transmit startup frames that fit into its own schedule. If this fails, or if clock correction signals an error, the node aborts the integration attempt and tries anew.

After receiving valid startup frames pairs for two consecutive double cycles from at least two coldstart nodes, the node leaves startup and enters operation. Thereby, it leaves startup at least two cycles after the node that initiated the coldstart. That means that all nodes of the cluster can leave startup at the end of cycle 7, just before entering cycle 8 (see Figure 7-10 and SDL macro INTEGRATION_CONSISTENCY_CHECK in Figure 7-19).

7.2.4.4 The *POC:coldstart listen* state

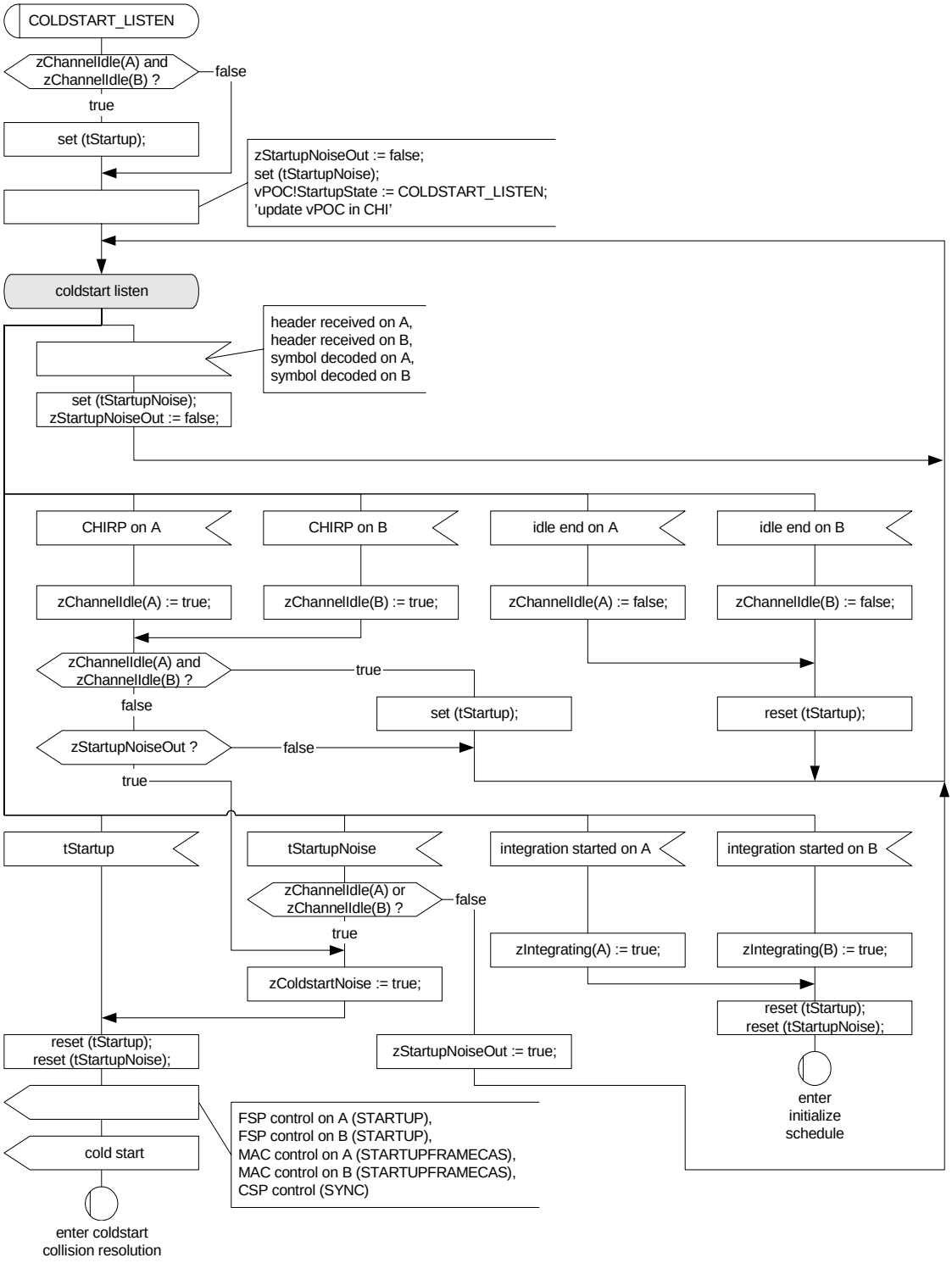

Figure 7-11: Transitions from the state *POC:coldstart listen* state [POC].[87]

[87] For a single channel node, the not-attached channel is assumed to be always idle.

A coldstart node still allowed[88] 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 the channel is noisy.

7.2.4.5 The *POC:coldstart collision resolution* state

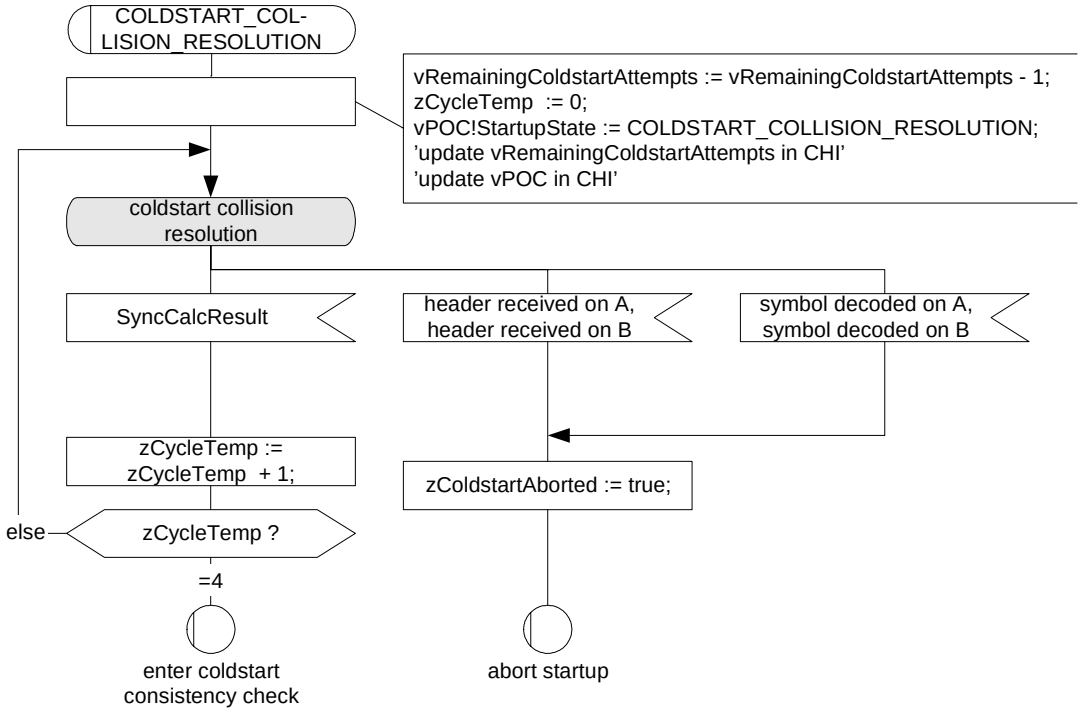

Figure 7-12: Transitions from the *POC:coldstart collision resolution* state [POC].

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.

[88] See macro STARTUP_PREPARE in Figure 7-9. The condition '*vRemainingColdStartAttempts* > 1' arises from the necessity of using up one round through the state *POC:coldstart collision resolution* for the collision resolution and needing the second round for actually integrating the other nodes.

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 state *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.

7.2.4.6 The *POC:coldstart consistency check* state

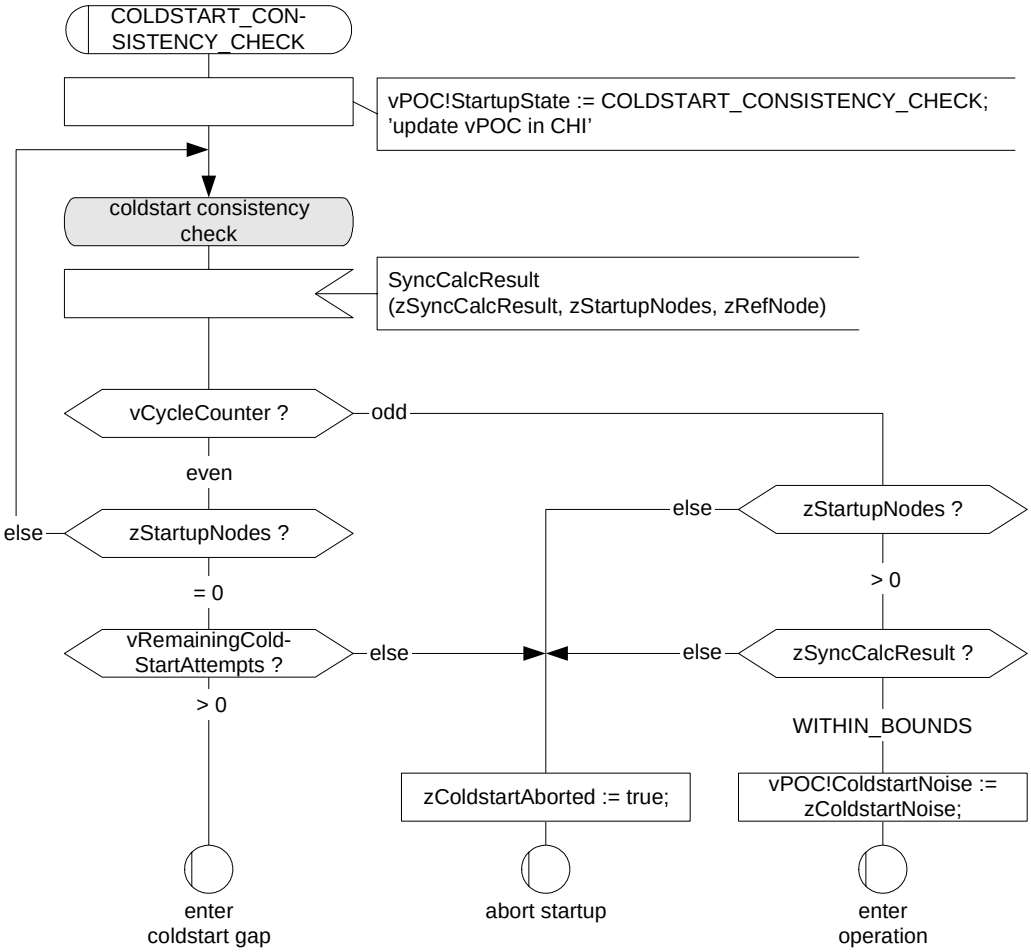

Figure 7-13: Transitions from the state *POC:coldstart consistency check* state [POC].

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 no valid startup frames are received 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 state *POC:coldstart gap* state to wait for the other coldstart nodes to get ready.

If a valid startup frame is received 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 valid pair of startup frames has been received and the clock correction signals no errors the node leaves startup and enters operation (see Chapter 2).

7.2.4.7 The *POC:coldstart gap* state

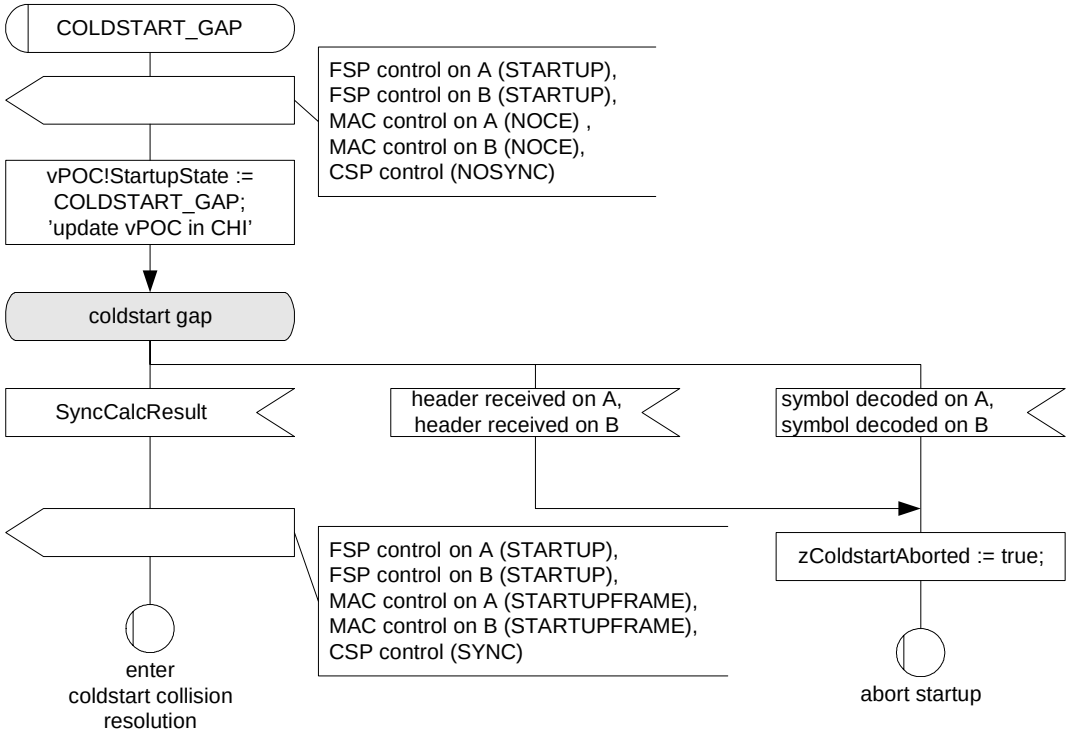

Figure 7-14: Transitions from the *POC:coldstart gap* state [POC].

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.

7.2.4.8 The *POC:initialize schedule* state

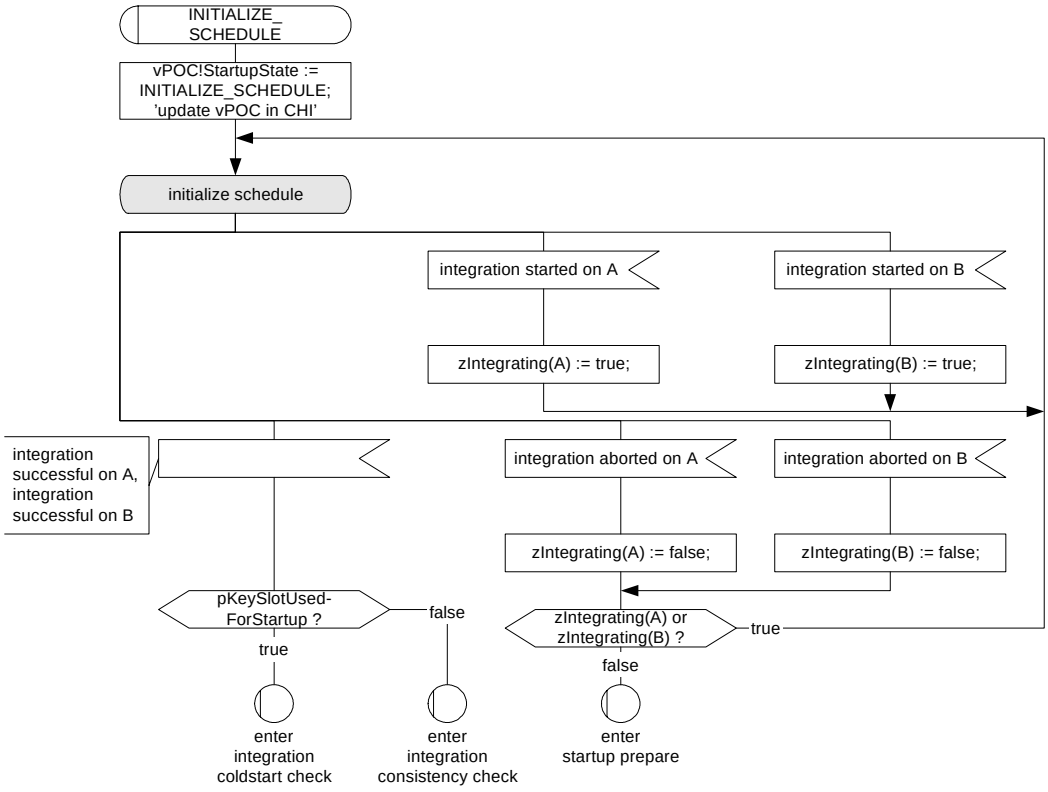

Figure 7-15: Transitions from the *POC:initialize schedule* state [POC].

As soon as a valid startup frame has been received in one of the listen states (see Figure 7-8), 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:integration consistency check* state is entered. Otherwise the node reenters the appropriate listen state.

7.2.4.9 The *POC:integration coldstart check* state

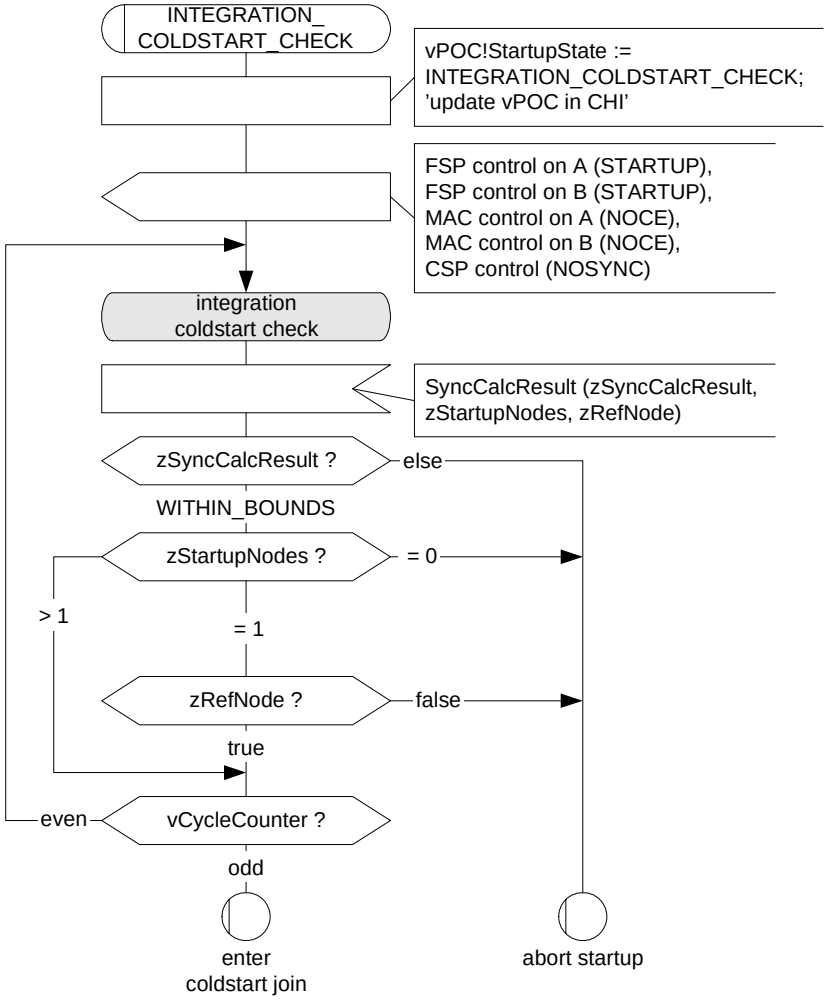

Figure 7-16: Transitions from the *POC:integration coldstart check* state [POC].

Only integrating (following) coldstart nodes pass through this state. In this state it shall be verified that the clock correction can be performed correctly and that the coldstart node that the node has initialized its schedule from is still available.

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.

7.2.4.10 The *POC:coldstart join* state

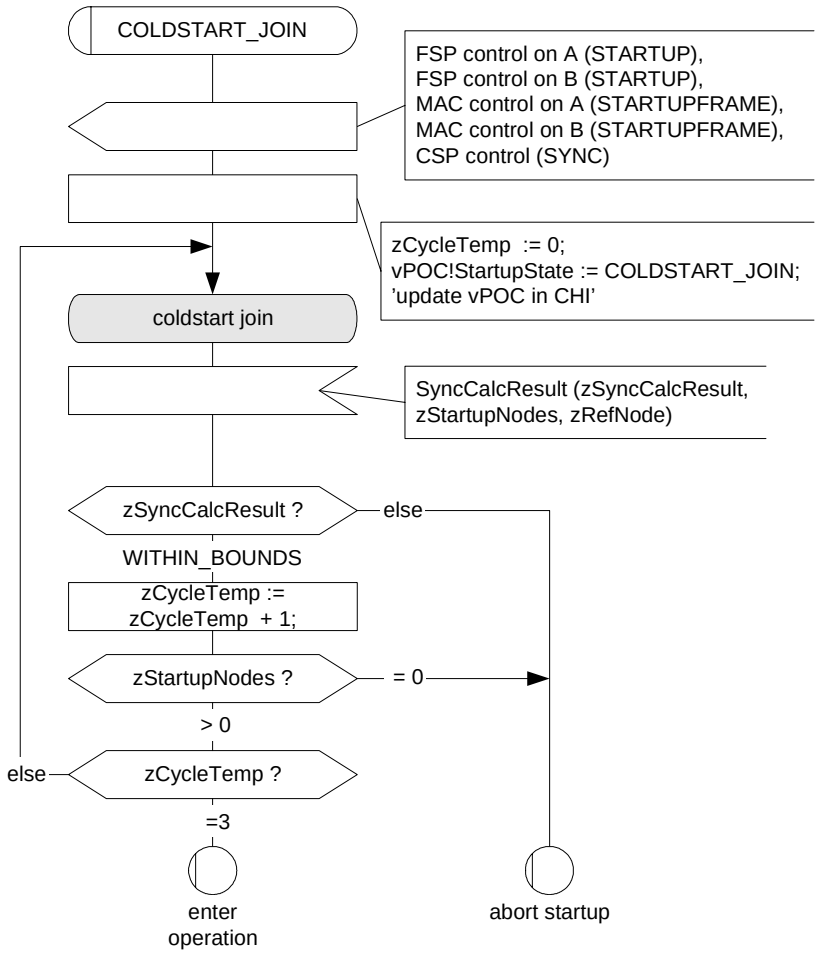

Figure 7-17: Transitions from the *POC:coldstart join* state [POC].

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).

7.2.4.11 The *POC:integration listen* state

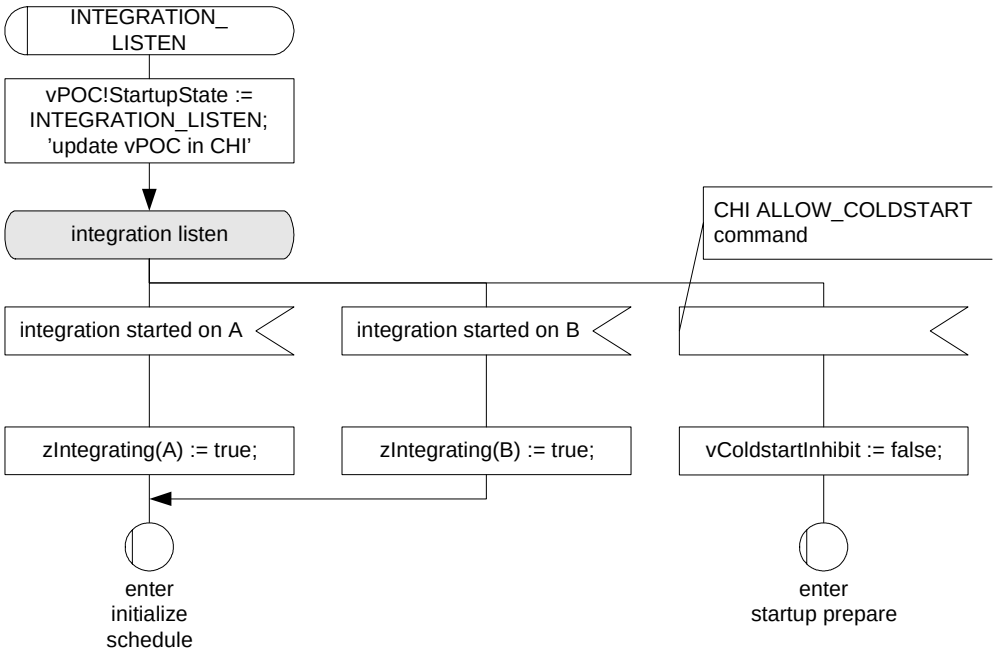

Figure 7-18: Transitions from the *POC:integration listen* state [POC].

In this state the node waits for either a valid startup frame or for the *vColdstartInhibit* flag to be cleared.

If the *vColdstartInhibit* flag is cleared the node reevaluates whether it is allowed to initiate a coldstart and consequently enter the state *POC:coldstart listen* state.

7.2.4.12 The *POC:integration consistency check* state

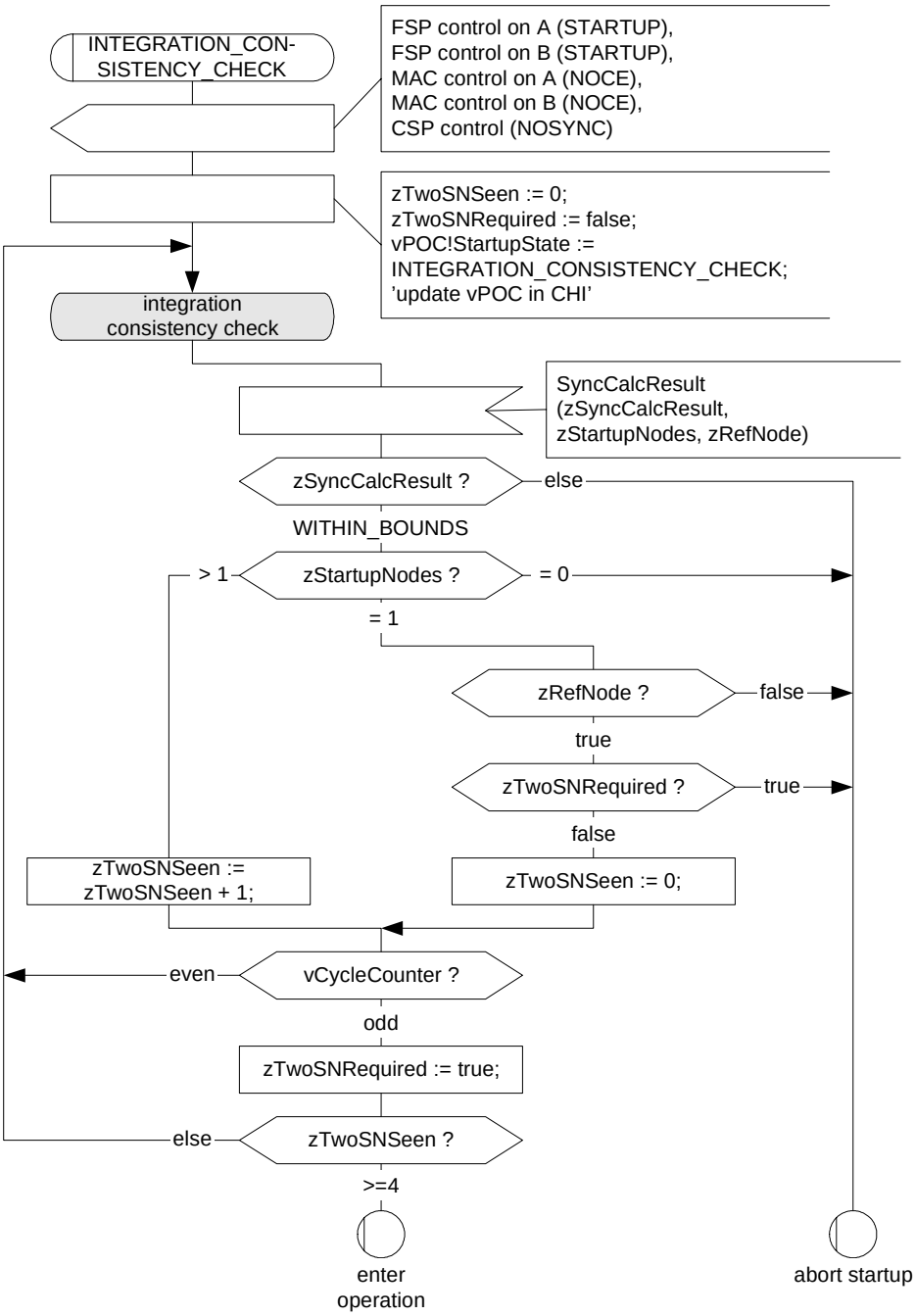

Figure 7-19: Transitions from the *POC:integration consistency check* state [POC].

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.

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.

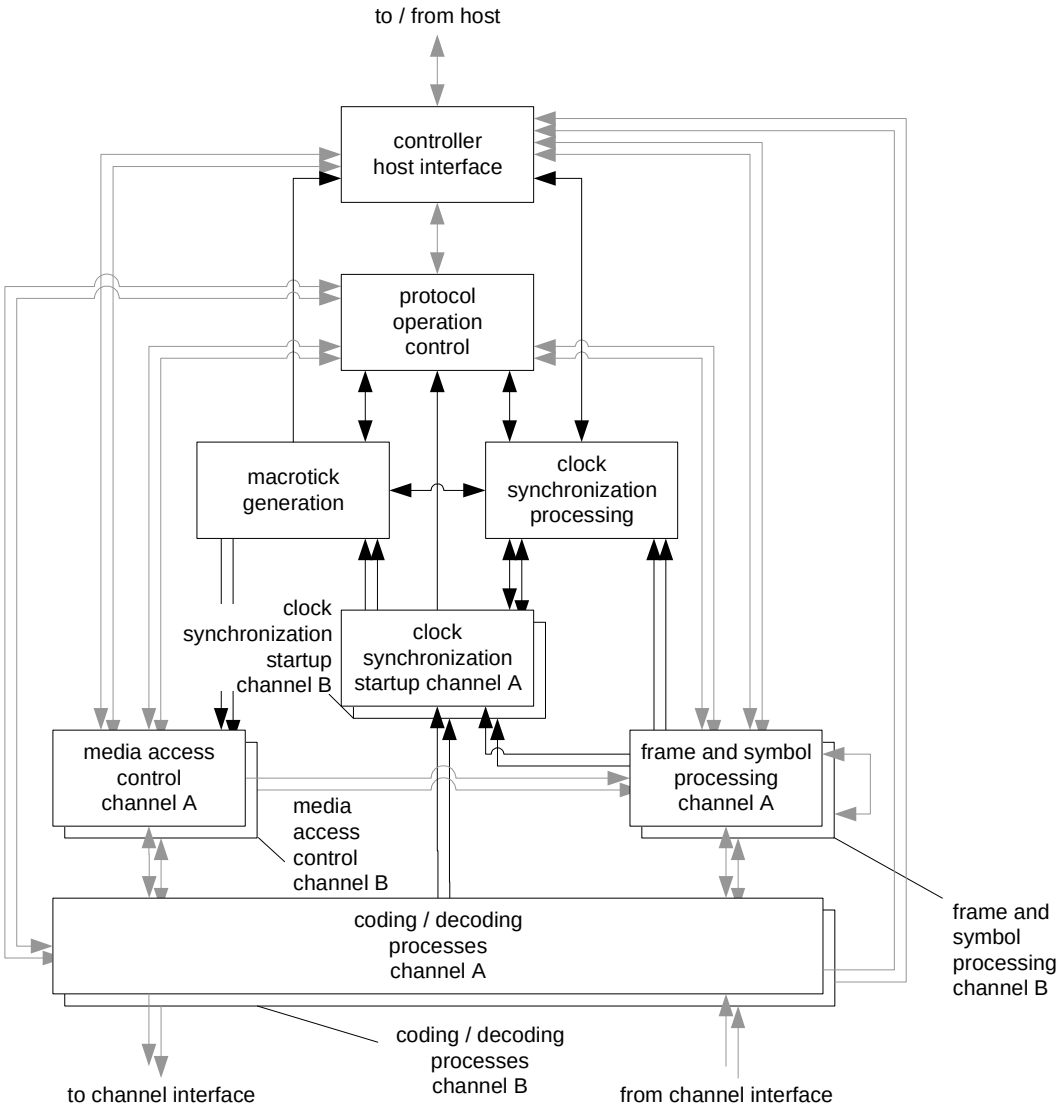

Figure 8-1: Clock synchronization context.

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[89].

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).

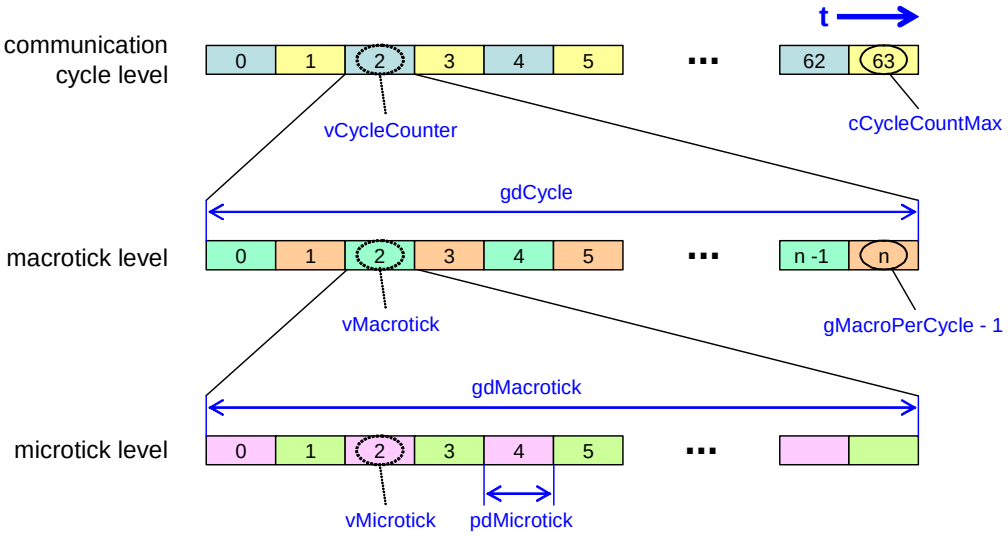

Figure 8-2: Timing hierarchy.

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).[90]

[89] 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).[91]

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*.[92]

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 *cCycleCountMax*. When *cCycleCountMax* 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:

- *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:

$$pMicroPerCycle = \text{round}(gdCycle \,/\, pdMicrotick)$$

[90] This is true even for the nominal (uncorrected) average duration of a macrotick (for example 6000 microticks distributed over 137 macroticks).
[91] The cycle number discrepancy is at most one, and lasts no longer than the precision of the system.
[92] 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.

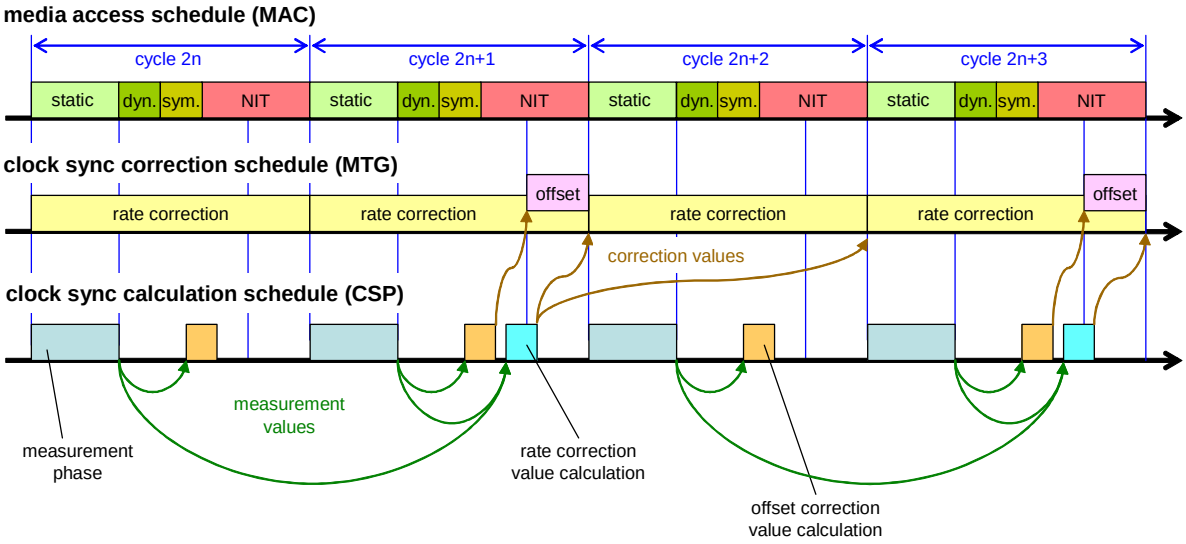

Figure 8-3: Timing relationship between clock synchronization, media access schedule, and the execution of clock synchronization functions.

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 *gOffsetCorrectionStart*, 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 *vOffsetCorrection*. *vOffsetCorrection* indicates the number of microticks that are added to the offset correction segment of the network idle time. *vOffsetCorrection* may be negative. The value of *vOffsetCorrection* is determined by the clock synchronization algorithm. The calculation of *vOffsetCorrection* takes place every cycle but a correction is only applied at the end of odd communication cycles. The calculation of *vOffsetCorrection* 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 *vRateCorrection*. *vRateCorrection* is an integer number of microticks that are added to the configured number of microticks in a communication cycle (*pMicroPerCycle*)[93]. *vRateCorrection* may be negative. The value of *vRateCorrection* is determined by the clock synchronization algorithm and is only computed once per double cycle. The calculation of *vRateCorrection* takes place following the static segment in an odd cycle. The calculation of *vRateCorrection* 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 8-4) 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 cold start 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 received sync frames.

[93] *pMicroPerCycle* is the configured number of microticks per communication cycle without correction.

```
syntype T_ArrayIndex = Integer
      constants 1 : 15
endsyntype;

syntype T_SyncNodes = Integer
      constants 0 : cSyncNodeMax
endsyntype;

newtype T_FrameIDTable
      Array(T_ArrayIndex, T_FrameID)
endnewtype;
```

Definition 8-4: Formal definition of T_ArrayIndex, T_SyncNodes, and T_FrameIDTable.

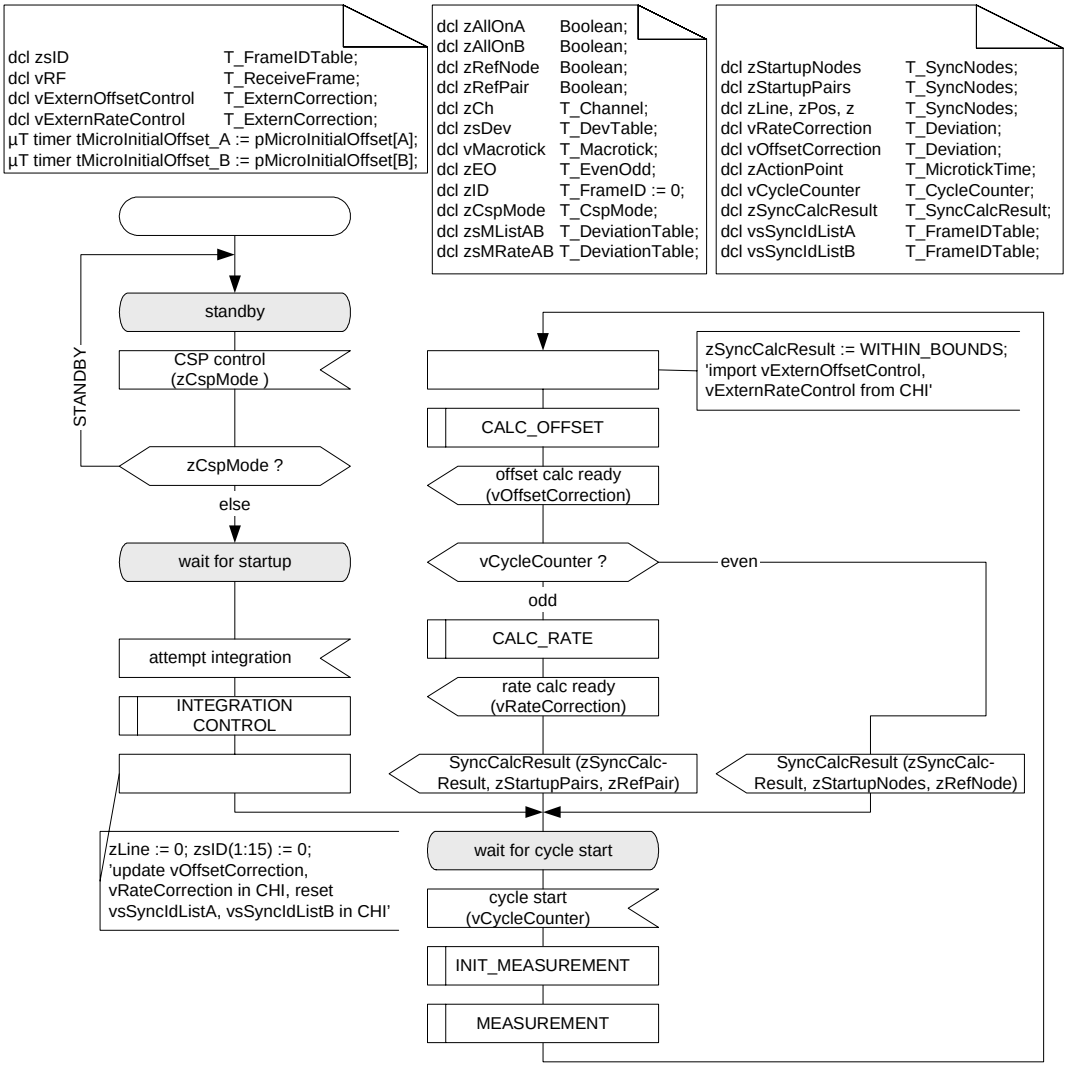

Figure 8-4: Clock synchronization process overview [CSP].

After finishing the startup procedure a repetitive sequence consisting of cycle initialization (Figure 8-10), a measurement phase (Figure 8-11), and offset (Figure 8-14) and rate (Figure 8-15) 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 8-5) handles mode changes done by the POC. It also handles process termination requests sent by the POC.

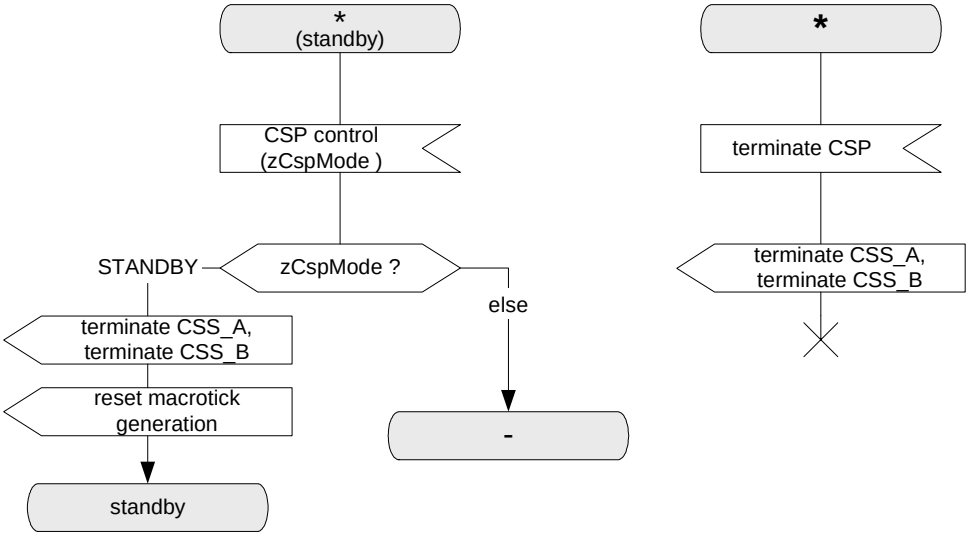

Figure 8-5: Clock synchronization control [CSP].

8.4 Startup of the clock

The startup of the node's internal synchronized clock 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 (synchronized) clock inside 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.

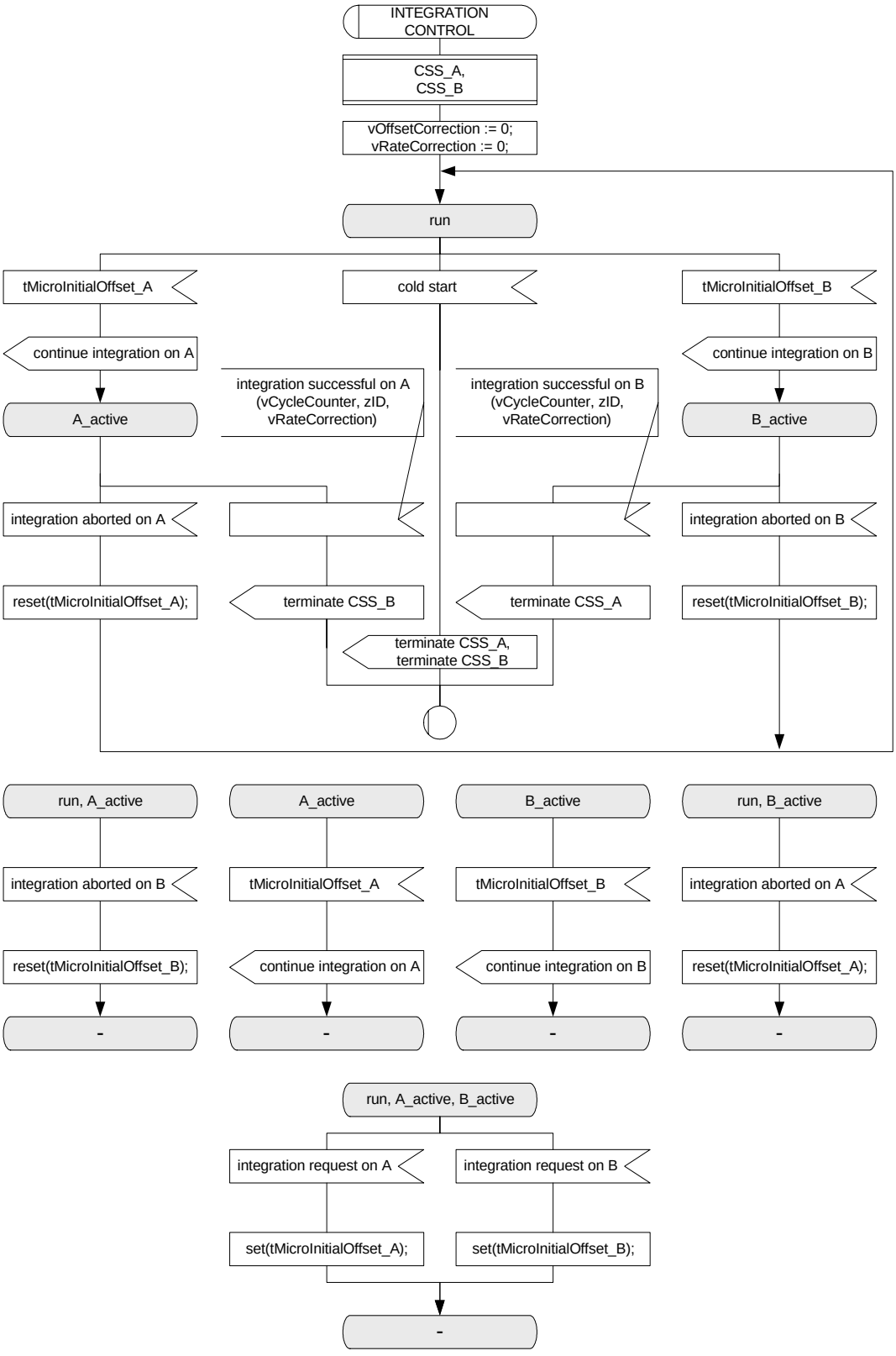

Figure 8-6: Integration control [CSP].

The startup procedure will be entered when the CSP receives the signal *attempt integration* from the POC (see Figure 8-4). The control of the node's startup is described in the INTEGRATION CONTROL macro depicted in Figure 8-6.

8.4.1 Cold start startup

If no ongoing communication on the channels is 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 8-17) 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 8-6) instantiates the clock synchronization startup processes for channel A and B (CSS_A, CSS_B).

After their instantiation, the CSS_A process (Figure 8-8) 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 8-17) 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).[94] 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.

[94] 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 8-8).

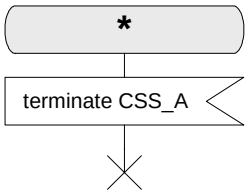

Figure 8-7: Clock synchronization startup control [CSS_A].

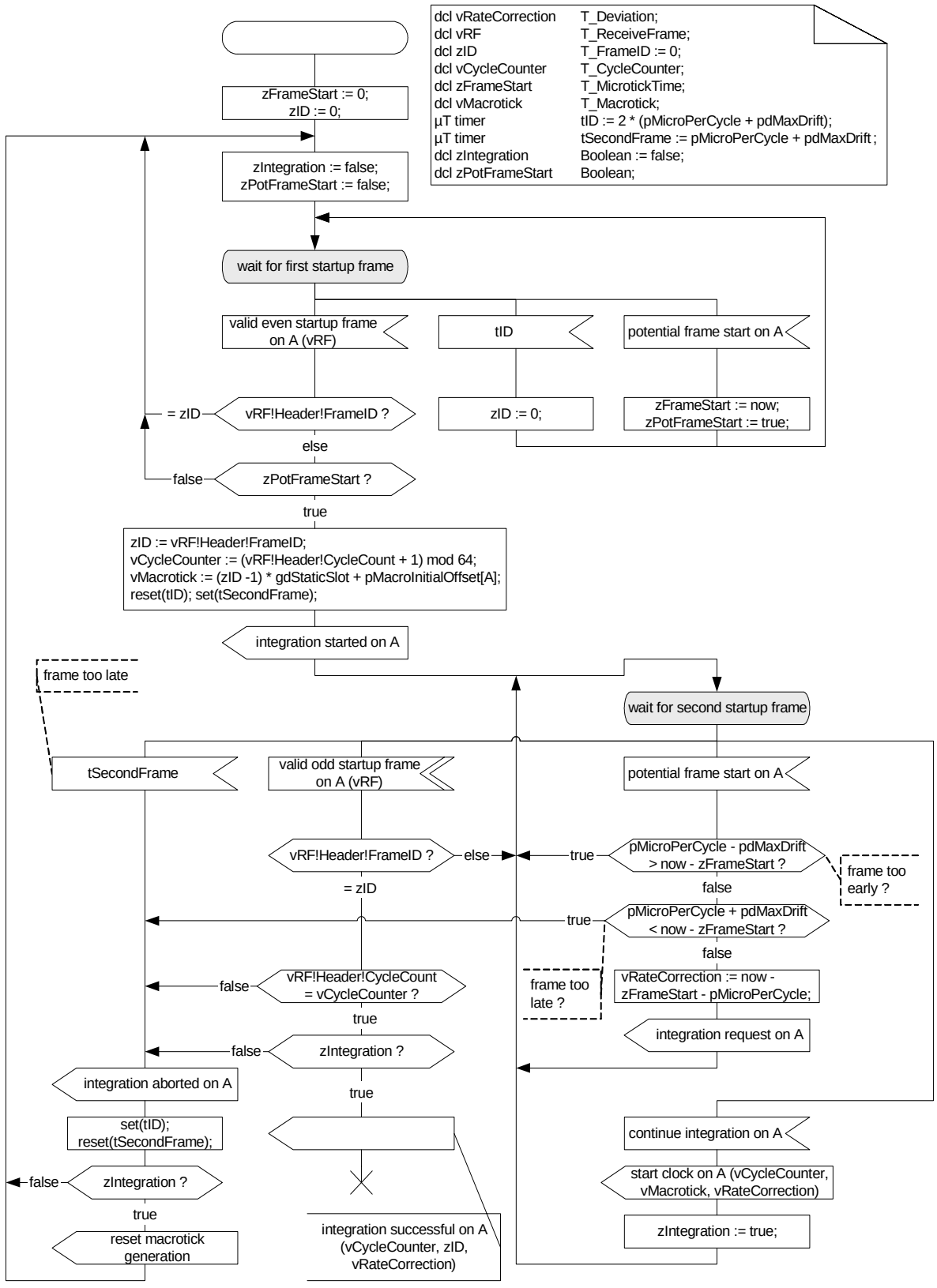

Figure 8-8: Clock synchronization startup process on channel A [CSS_A][95].

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 8-6). This behavior of this termination is depicted in Figure 8-7.

The timer *tSecondFrame* in Figure 8-8 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).

[95] The priority input symbol on the state *CSS_A:wait for second startup frame* 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.

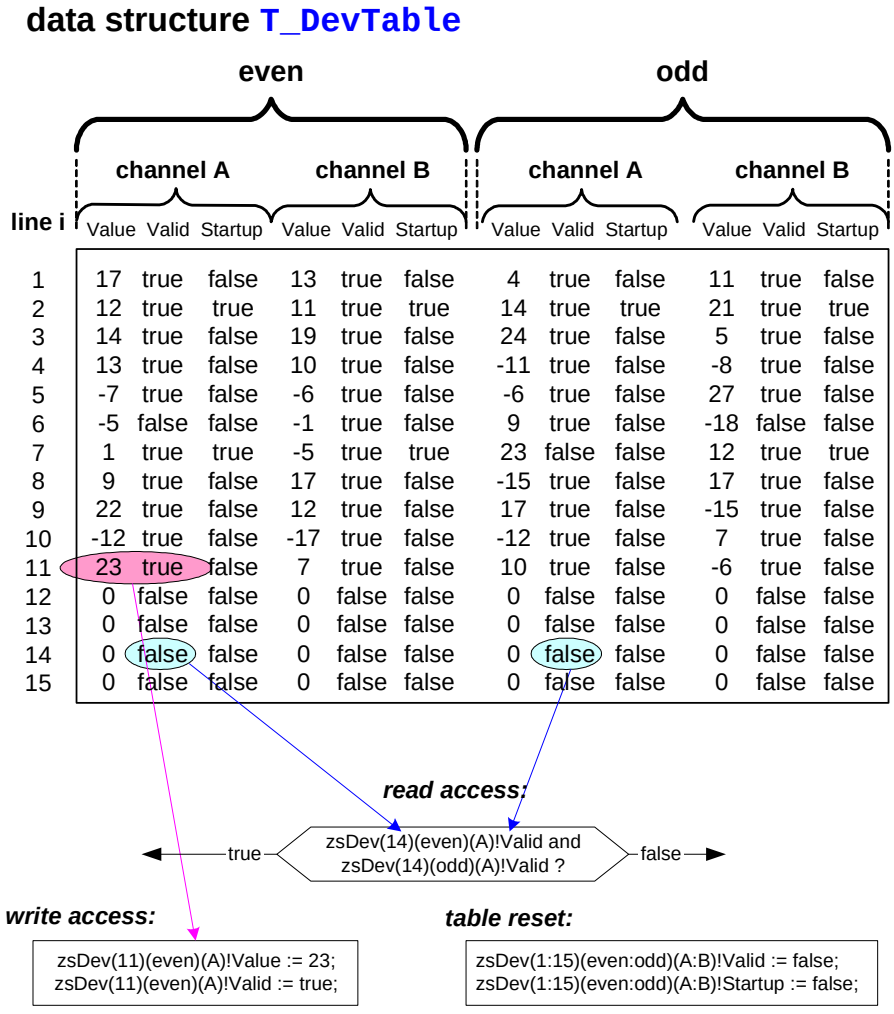

Figure 8-9: Data structure example used in the following explanations.

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 8-5: 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 8-6: 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 node. 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. 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 8-9 gives an example of this data structure.

8.5.2 Initialization

The data structure introduced in section 8.5.1 is used to instantiate a variable (*zsDev*). A portion of this variable will be reset at the beginning of every communication cycle; the even part at the beginning of an even cycle and the odd part at the beginning of an odd cycle. Additionally, if the node is configured to transmit sync frames (mode SYNC), corresponding entries are stored in the variable as depicted in Figure 8-10.

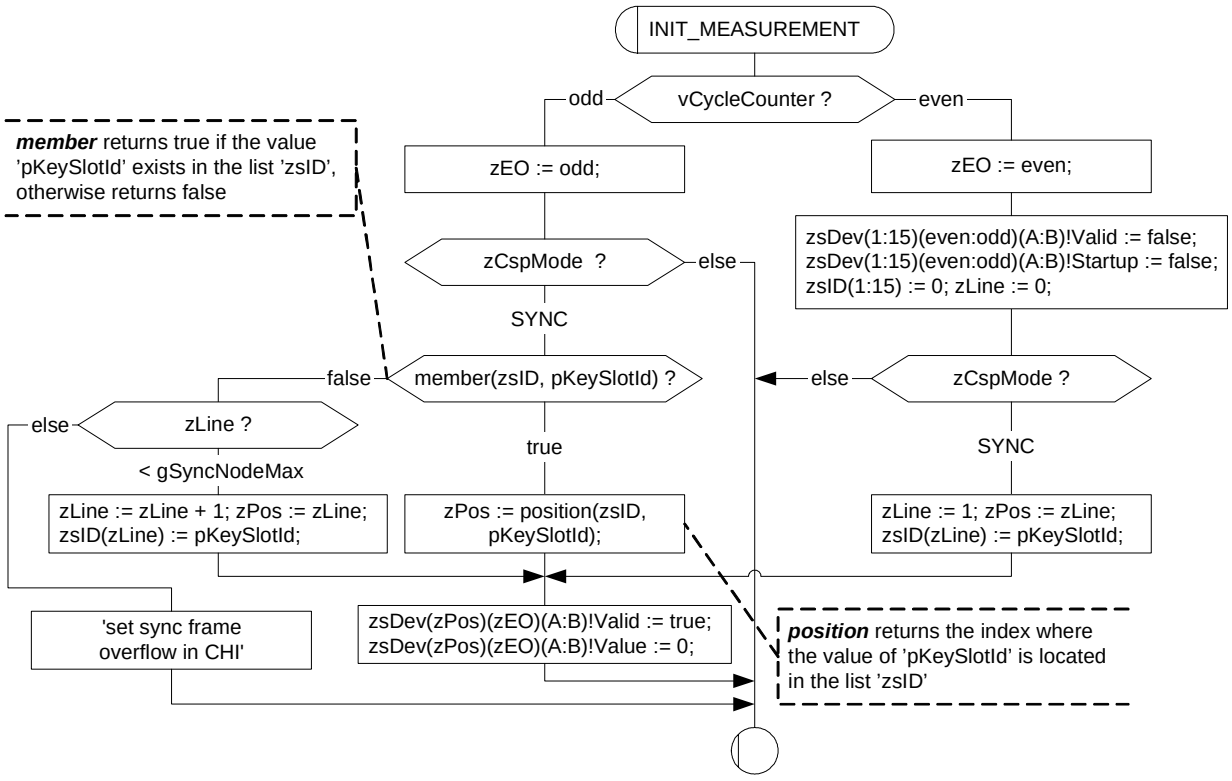

Figure 8-10: Initialization of the data structure for measurement [CSP].

8.5.3 Time measurement storage

The expected arrival time of a 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 3.2.5.

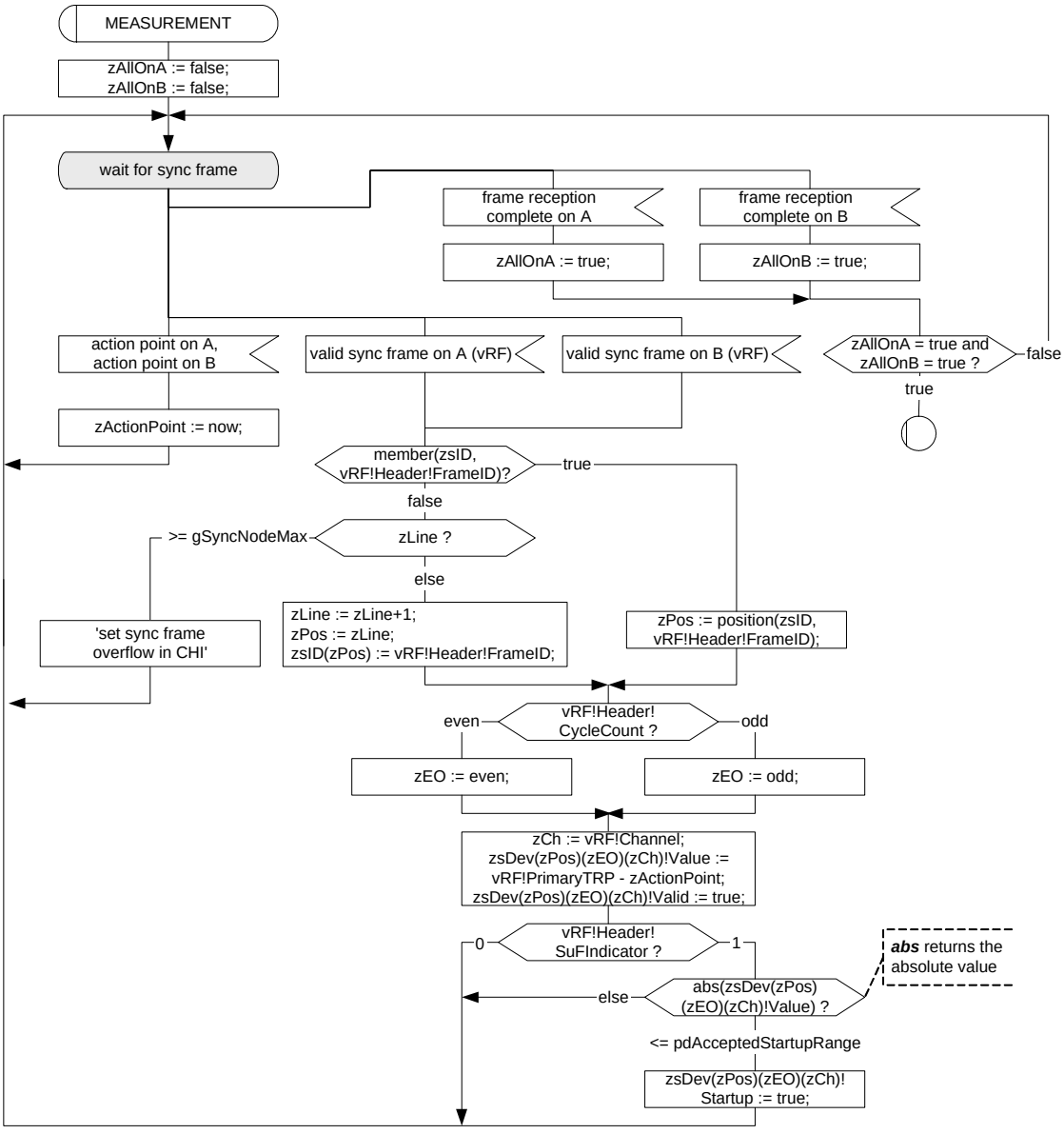

Figure 8-11: Measurement and storage of the deviation values [CSP].

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 8-11). The measurement phase ends when the static segment ends.

The reception of more than *gSyncNodeMax* sync frames per channel in one communication cycle indicates an error inside the cluster. This is reported to the host and only the first *gSyncNodeMax* received sync frames are used for the correction value calculation.

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). The algorithm works as follows (see Figure 8-12 and Figure 8-13):

1. The algorithm determines the value of a parameter, k, based on the number of values in the sorted list (see Table 8-1).[96]

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.[97] The resulting value is assumed to represent the node's deviation from the global time base and serves as the correction term.

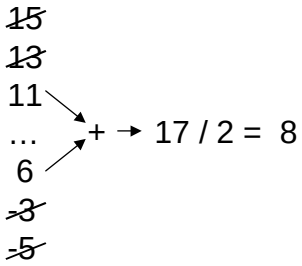

Figure 8-12: Algorithm for clock correction value calculation (k=2).

[96] The parameter k is not the number of asymmetric faults that can be tolerated.
[97] The division by two of odd numbers should truncate toward zero such that the result is an integer number. Example: 17 / 2 = 8 and -17 / 2 = -8.

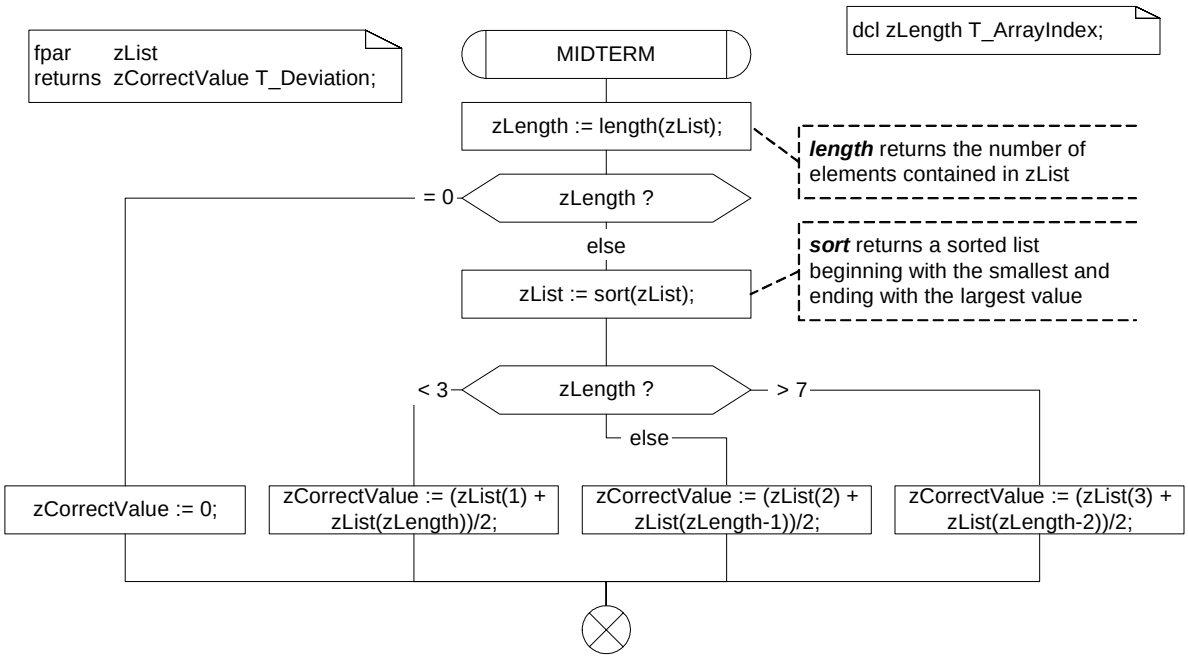

Figure 8-13: Fault Tolerant Midpoint Procedure.

8.6.2 Calculation of the offset correction value

The offset correction value *vOffsetCorrection* 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 8-14 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 8-14:

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 no sync frames were received an error flag is set to true.
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 an error flag is set to true 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:

```
newtype T_DeviationTable
      Array(T_ArrayIndex, T_Deviation)
endnewtype;
```

Definition 8-7: Formal definition of T_DeviationTable.

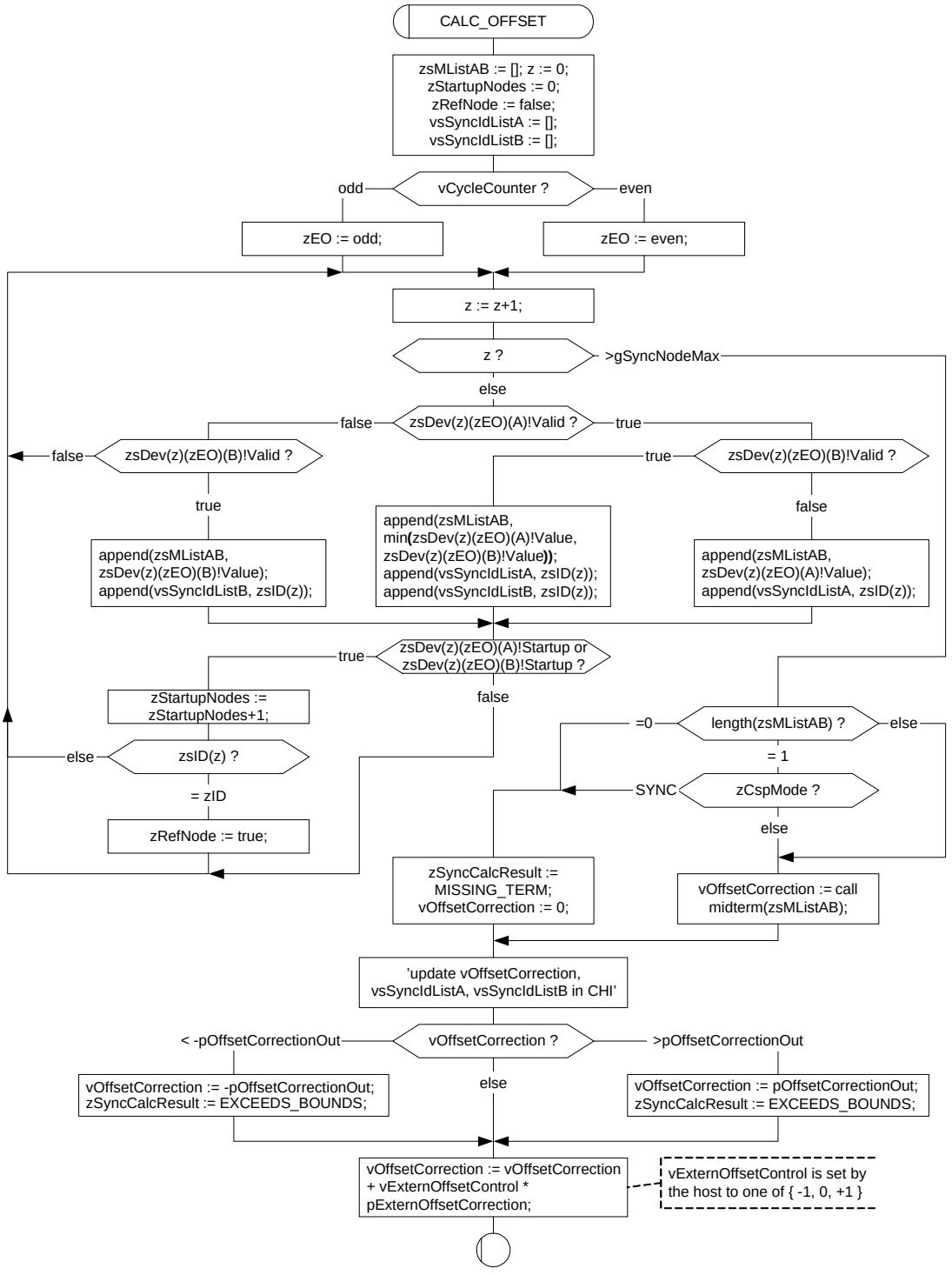

Figure 8-14: Calculation of the offset correction value [CSP].

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 8-4 (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 8-15.

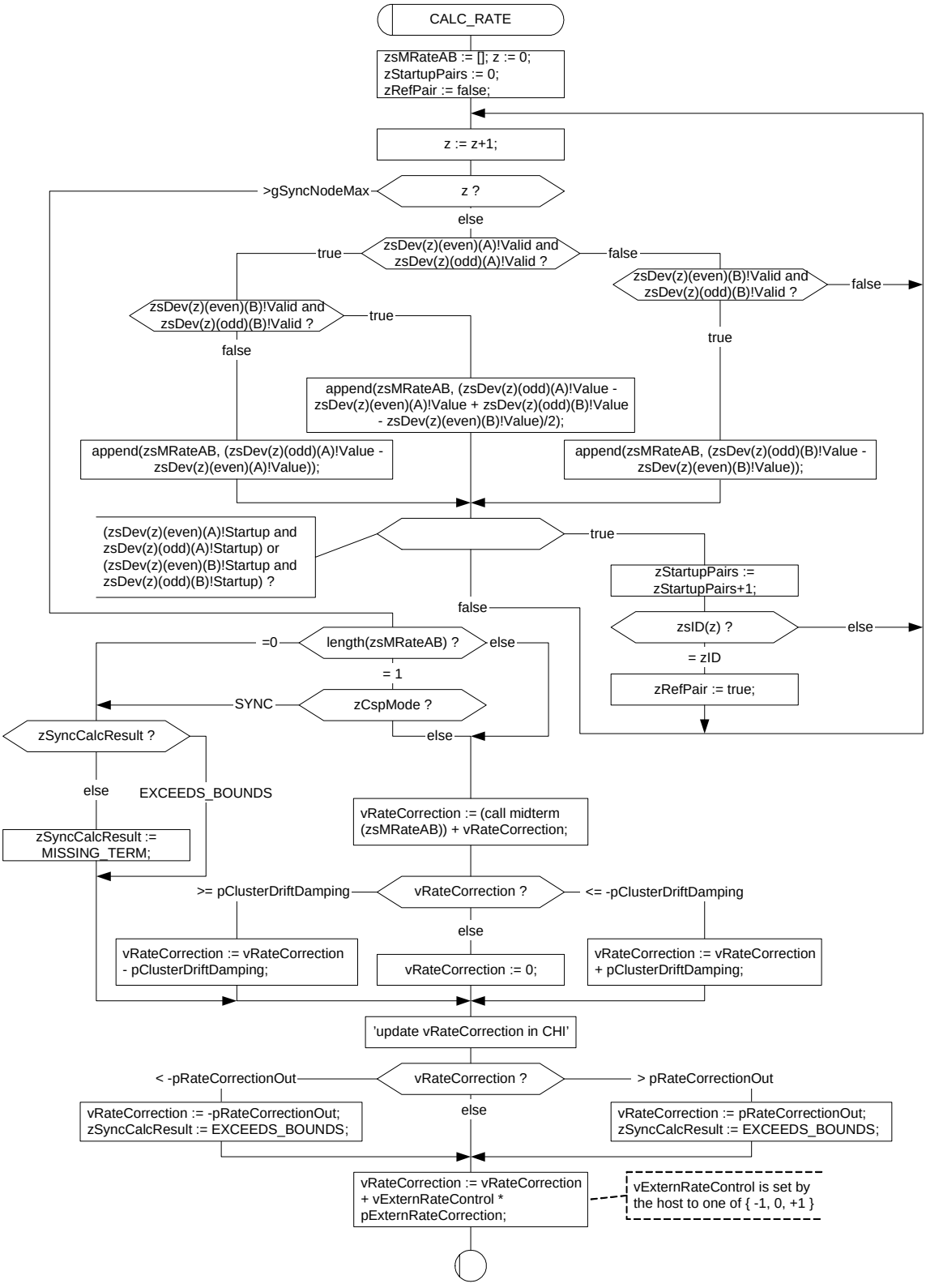

Figure 8-15: Calculation of the rate correction value [CSP].[98]

The rate correction value *vRateCorrection* 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 8-15:

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 the same slot, 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 no sync frame pairs were received an error flag is set to true.
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. The correction term is checked against specified limits. If the correction term exceeds the specified limits an error flag is set to true and the correction term is set to the maximum or minimum value as appropriate (see section 8.6.4).
6. If appropriate, an external correction value supplied by the host is added to the calculated correction term (see section 8.6.5).

The *pClusterDriftDamping* parameter should be configured in such a way that the damping value in all nodes has approximately the same duration.[99]

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 8-14 and Figure 8-15)

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* <= *vRateCorrection* <= *pRateCorrectionOut*

- *pOffsetCorrectionOut* <= *vOffsetCorrection* <= *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.

[98] The division by two of odd numbers should truncate toward zero such that the result is an integer number. Example: 17 / 2 = 8 and -17 / 2 = -8.

[99] A node-specific configuration value is used to allow clusters that have different microtick durations in different nodes.

8.6.5 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-determined 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 states.

The size of the external rate and the external offset correction values *pExternOffsetCorrection* and *pExternRateCorrection* 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 8-8: Formal definition of T_ExternCorrection.

The handling of the external correction values is depicted in Figure 8-14 and Figure 8-15.

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.

8.7 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 8-17) generates (corrected) macroticks. There are two different ways to start the macrotick generation process:

- The protocol operation control (POC) process starts the MTG process (*cold start* signal) if the conditions to start the node as a coldstart node are satisfied, or
- The node is an integrating node (*start clock on A* or *start clock on B* signal) and the integration of the node was successful (see clock synchronization startup).

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.

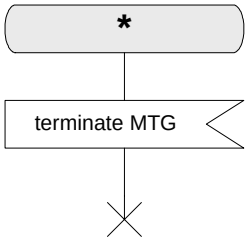

Figure 8-16: Termination of the macrotick generation process [MTG].

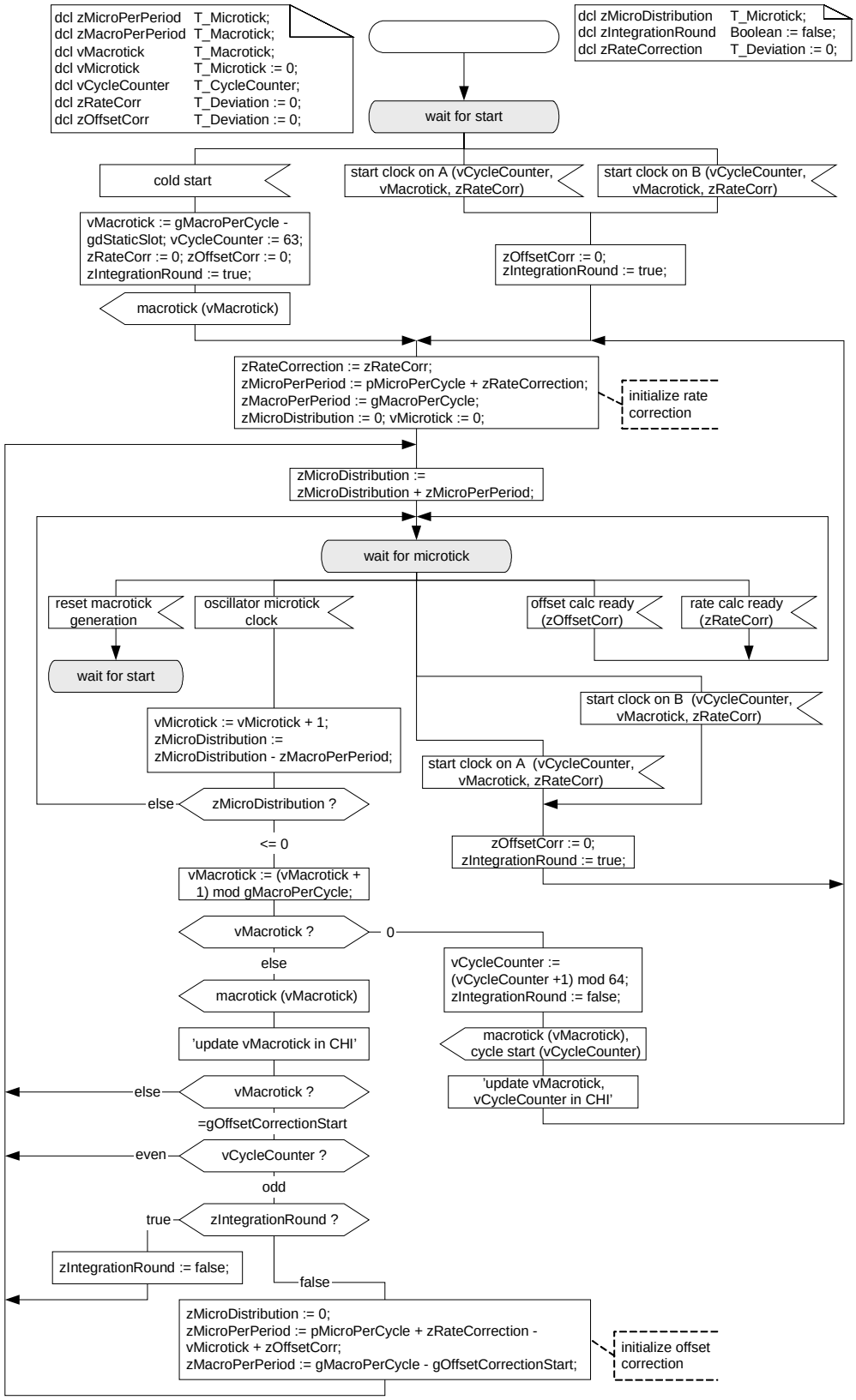

Figure 8-17: Macrotick generation [MTG].

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
Number of sync nodes	[3[a] … *cSyncNodeMax*]
Number of static slots	>= number of sync nodes
Number of sync frames	= number of sync nodes

Table 8-3: Configuration rules for the distributed clock synchronization.

[a] Even if only two sync nodes are operating in *POC:normal active* state communication within the cluster will still take place in accordance with the communication cycle. The communication in such a cluster will, however, stop upon failure of any one of the two sync nodes.

A number of nodes must be configured as *sync nodes* depending on the following rules:

- At least three nodes shall be configured to be sync nodes.
- At most *cSyncNodeMax* 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, one on each channel, in a single slot of the static segment.[100] Sync nodes that only support a single channel shall send one sync frame in one slot of the static segment. The sync frames shall be sent in the same slot in each cycle.
- Non sync nodes must not transmit frames with the sync frame indicator set to one.

[100] The frames sent on the two channels in the sync slot do not need to be identical (i.e., they may have differing payloads), but they must both 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.[101]

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.

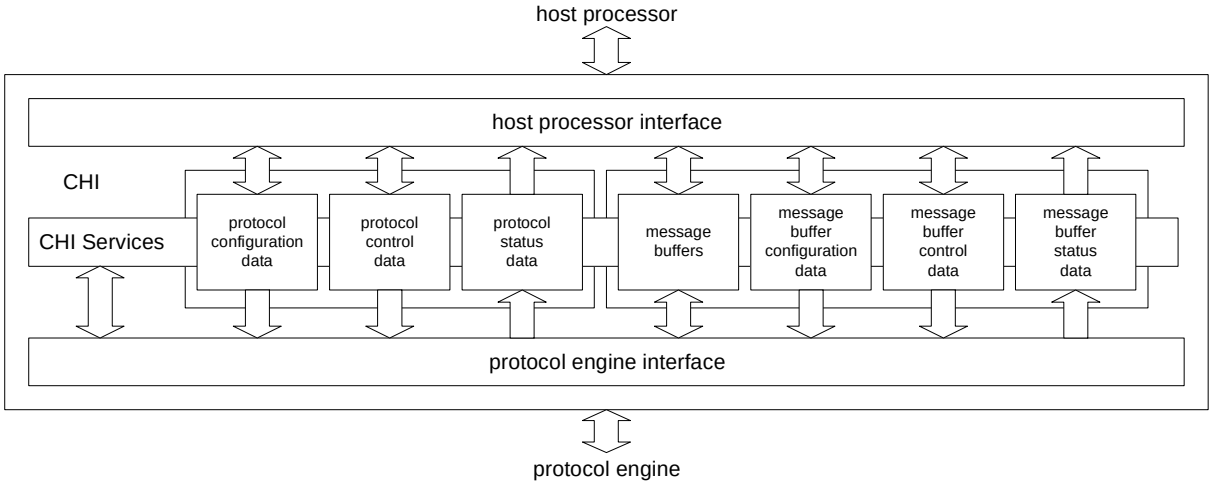

Figure 9-1: Conceptual architecture of the controller host interface.

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, the message buffer control 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.

9.2 Description

The relationships between the CHI and the other protocol processes are depicted in Figure 9-2[102].

[101] 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.

[102] 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.

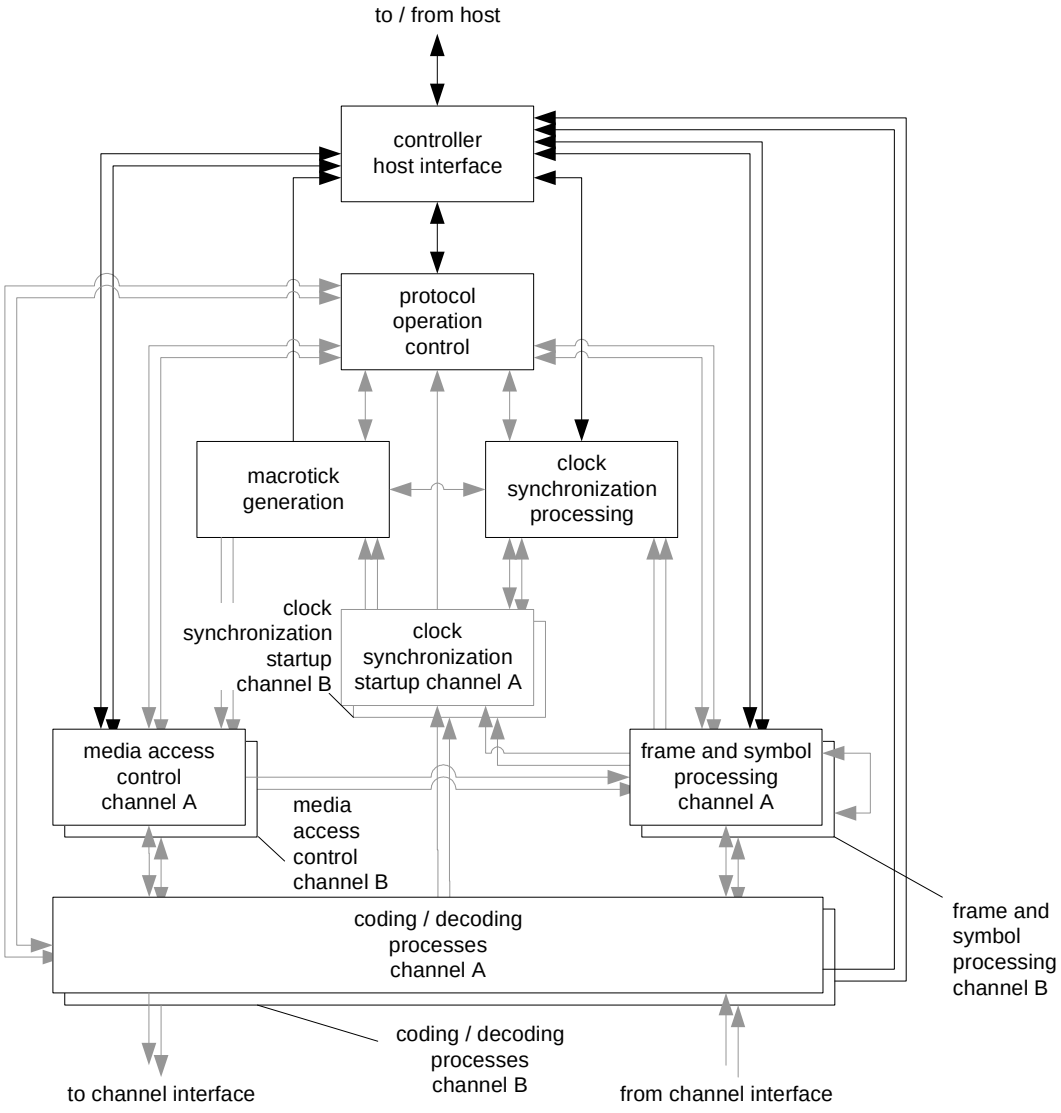

Figure 9-2: Controller host interface context.

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.

9.3.1.1.1 Communication cycle timing-related protocol configuration data

All configuration data relating to the following communication cycle-related protocol 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 and symbol window,
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.

9.3.1.1.2 Protocol operation-related protocol configuration data

All configuration data relating to the following protocol operation-related protocol parameters shall be treated as protocol configuration data:

1. the enumeration *pChannels* that indicates the channels to which the node is connected,
2. the maximum number of times *gColdStartAttempts* that a node is permitted to attempt to start the cluster by initiating schedule synchronization,
3. the upper limit *gListenNoise* for the start up and wake up listen timeout in the presence of noise,
4. the enumeration *pWakeupChannel* that indicates on which channel a wakeup symbol shall be sent upon issuing the WAKEUP command.
5. 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* into *POC:halt* state,
6. 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,
7. the number of microticks *pdAcceptedStartupRange* constituting the expanded range of measured deviation for startup frames during integration,
8. the number of microticks *pdListenTimeout* constituting the upper limit for the startup and wakeup listen,
9. the number of microticks *pClusterDriftDamping* constituting the cluster drift damping factor used for rate correction within clock synchronization,
10. 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,
11. 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,
12. the number of microticks *pdMaxDrift* that constitute the maximum drift offset between two nodes operating with unsynchronized clocks for one communication cycle,
13. the number of macroticks *gOffsetCorrectionStart* between the start of the communication cycle and the start of the offset correction within the NIT,
14. the number of microticks *pMicroInitialOffset[A]* between the actual time reference point on channel A and the subsequent macrotick boundary of the secondary time reference point,

15. the number of microticks *pMicroInitialOffset[B]* between the actual time reference point on channel B and the subsequent macrotick boundary of the secondary time reference point,
16. the number of microticks *pExternOffsetCorrection* constituting the correction term used to correct the calculated offset correction value in the course of external clock synchronization,
17. the number of microticks *pExternRateCorrection* constituting the correction term used to correct the calculated rate correction value in the course of external clock synchronization,
18. the number of microticks *pOffsetCorrectionOut* constituting the upper bound for a permissible offset correction,
19. the number of microticks *pRateCorrectionOut* constituting the upper bound for a permissible rate correction,
20. the Boolean flag *pAllowHaltDueToClock* that controls the transition to the *POC:halt* state due to a clock synchronization error,
21. 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,
22. the Boolean flag *pSingleSlotEnabled* that defines whether a node will send in only one or in all configured slots after completing startup,
23. the Boolean flag *pKeySlotUsedForStartup* that defines whether the specific node shall send startup frames,
24. the Boolean flag *pKeySlotUsedForSync* that defines whether the specific node shall send sync frames,
25. the number of the static slot *pKeySlotId* in which a startup frame, a sync frame or a normal frame shall be sent in accordance with *pKeySlotUsedForStartup*, *pKeySlotUsedForSync* and *pSingleSlotEnabled*,
26. the number of microticks *pDecodingCorrection* used by the node to calculate the primary time reference point.

9.3.1.1.3 Frame-related protocol configuration data

All configuration data relating to the following frame-related protocol parameters shall be treated as protocol configuration data:

1. the number of two-byte words *gPayloadLengthStatic* contained in the payload segment of a static frame,
2. the number of bits *gdTSSTransmitter* within the transmission start sequence.

9.3.1.1.4 Symbol-related protocol configuration data

All configuration data relating to the following symbol-related protocol parameters shall be treated as protocol configuration data:

1. the number of bits *gdWakeupSymbolRxIdle* used by the node to test the duration of the received 'idle' part of the wakeup symbol,
2. the number of bits *gdWakeupSymbolRxLow* used by the node to test the duration of the received 'active low' part of the wakeup symbol,
3. the number of bits *gdWakeupSymbolRxWindow* used by the node to test the duration of the received wakeup symbol,
4. the number of bits *gdWakeupSymbolTxIdle* used by the node to transmit the 'idle' part of the wakeup symbol,
5. the number of bits *gdWakeupSymbolTxLow* used by the node to transmit the 'active low' part of the wakeup symbol,
6. the number of wakeup symbols *pWakeupPattern* to be sent by the node.

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 2-2:[103]

1. an ALLOW_COLDSTART command that activates the capability of the node to cold start the cluster,
2. an ALL_SLOTS command that controls the transition from the single slot transmission mode to the all slots transmission mode,
3. a CONFIG command that causes the 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 the transition of the POC process from the *POC:config* state to the *POC:ready* state,
5. a FREEZE command that causes the transition of the POC process from any POC state to the *POC:halt* state,
6. a READY command that causes the 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 the transition of the POC process from the *POC:halt* state to the *POC:default config* state,
9. a HALT command that causes the transition of the POC process from the *POC:normal active* or *POC:normal passive* states to the *POC:halt* state,
10. a WAKEUP command that initiates the wakeup procedure.

9.3.1.2.2 Control of MTS transmission

The CHI shall provide means for the host

1. to request a transmission of an MTS on channel A within the symbol window of channel A (as defined within the MAC_A process),
2. to request a transmission of an MTS on channel B within the symbol window of channel B (as defined within the MAC_B process).

9.3.1.2.3 Control of external clock synchronization

The CHI shall provide means for the host

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*.

9.3.1.3 Protocol status data

9.3.1.3.1 Protocol operation control-related status data

The following protocol operation control-related 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!CHIHaltRequest* (as maintained by the POC process),

[103] 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.

4. the flag *vPOC!ColdstartNoise* (as maintained by the POC process),
5. the status variable *vPOC!SlotMode* (as maintained by the POC process),
6. the status variable *vPOC!ErrorMode* (as maintained by the POC process),
7. the number of consecutive even/odd cycle pairs *vAllowPassiveToActive* that have passed with valid rate and offset correction terms, but 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),
8. the status variable *vPOC!WakeupStatus* (as maintained by the POC process),
9. the status variable *vPOC!StartupState* (as maintained by the POC process).

9.3.1.3.2 Startup-related status data

The following startup-related 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 startup-related events:

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.

9.3.1.3.3 Time-related status data

The following time-related protocol 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 time-related protocol variables shall be provided in the CHI:

1. the rate correction value *vRateCorrection* (in accordance with the CSP process),
2. the offset correction value *vOffsetCorrection* (as maintained by the CSP process).

The CHI shall provide an indicator for the following time-related event:

1. a sync frame overflow indicator that shall be set upon 'set sync frame overflow 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 in CHI' (in accordance with the MAC_A process),
3. a *pLatestTx* violation status indicator for channel B that shall be set upon 'set *pLatestTx* violation status indicator in CHI' (in accordance with the MAC_B process),
4. a transmission across boundary violation status indicator for channel A that shall be set upon 'set transmission across slot boundary violation indicator 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 in CHI' (in accordance with the FSP_B process).

9.3.1.3.4 Synchronization frame-related status data

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-related protocol 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 Symbol window-related status data

A snapshot of the following symbol window-related protocol variables shall be provided in the CHI for the slot status *vSS* established at the end of the symbol window for each channel:

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).

Table 9-1 lists all possible combinations of *ValidMTS*, *SyntaxError* and *BViolation* for the symbol window along with a set of interpretations for one channel.

ValidMTS	Syntax Error	BViolation	Nothing was received (silence)	A syntactically valid symbol (MTS) was received	Additional activity and a syntactically valid symbol (MTS) was received
false	false	false	Yes	No	-
false	true	false	No	No	-
false	false	true	No	No	-
false	true	true	No	No	-
true	false	false	No	Yes	No
true	true	false	No	Yes	Yes
true	false	true	No	Yes	Yes
true	true	true	No	Yes	Yes

Table 9-1: Symbol window status interpretation.

9.3.1.3.6 NIT-related status data

A snapshot of the following NIT-related protocol variables shall be provided in the CHI for the slot status *vSS* established at the end of the NIT for each channel:

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).

9.3.1.3.7 Aggregated channel status-related status data

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 data is aggregated over a period that is freely definable by the host.

The CHI shall provide indicators for the following channel activity-related events:

1. a channel specific valid frame indicator for channel A that shall be set if one or more valid frames were 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 one or more valid frames were 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 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 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.

9.3.1.3.8 Wakeup-related status data

The CHI shall provide indicators for the following wakeup-related events:

1. a wakeup pattern received indicator for channel A that shall be set upon 'set WUP received indicator on A in CHI' (in accordance with the WUPDEC_A process) and reset under control of the host,
2. a wakeup pattern received indicator for channel B that shall be set upon 'set WUP received indicator on B in CHI' (in accordance with the WUPDEC_B process) and reset under control of the host.

9.3.1.3.9 Dynamic segment-related status data

The following dynamic segment-related status variables shall be provided in the CHI:

1. the value of the channel dependent *vSlotCounter[A]* at the time of the last frame transmission by the node on channel A in the dynamic segment as reflected by the variable *zLastDynTxSlot* 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,[104]
2. the value of the channel dependent *vSlotCounter[B]* at the time of the last frame transmission by the node on channel B in the dynamic segment as reflected by the variable *zLastDynTxSlot* 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.

9.3.2 Message data interface

The message data interface addresses the exchange of message data between the host and the FlexRay communication protocol within each node. Message transmission pertains to the message data flow from the host to the FlexRay communication protocol and message reception pertains to the message data flow from the FlexRay communication protocol to the host.

Message transmission as well as message reception has associated message buffer configuration data, message buffer control data, and message buffer status data.

The host shall have read access to message buffer configuration data independently of the protocol state. Host write access to the buffer configuration shall be allowed in the *POC:config* state, and may be allowed in other POC states.[105]

9.3.2.1 Message transmission

Message transmission operates on non-queued transmit buffers. A non-queued transmit buffer is a data storage structure

1. for which the host has access to the data through both a read operation and a write operation,
2. for which the protocol accesses the data through a read operation, and
3. in which new values overwrite former values.

9.3.2.1.1 Transmission slot assignment

A specific TDMA slot in either the static or the dynamic segment shall be assigned to a node by assigning the corresponding slot ID to the node according to the constraints listed in Chapter 5.

Nodes own a list of assigned slots for each channel. With the exception of the sync slot or the single slot used for single slot mode, the list of assigned slots may be modified during the operation of the cluster. The assignment of a sync slot / single slot can only be changed during *POC:config*.

[104] This 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.

[105] It is not required that any implementation supports the modification of buffer configuration data in other POC states than the *POC:config* state.

In the static segment

When a slot occurs, if the slot is assigned to a node on a channel that node must transmit either a normal frame or a null frame on that channel. Specifically, a null frame will be sent if there is no data ready, or if there is no match on a transmit filter (cycle counter filtering, for example).

In the dynamic segment

When a slot occurs, if a slot is assigned to a node on a channel that node only transmits a frame on that channel if there is data ready and there is a match on relevant transmit filters (no null frames are sent).

In both segments

The system designer must ensure that in any given cycle no two nodes transmit in the same slot on the same channel.

It is the intention of the protocol that in the static segment each slot on a channel is owned by exactly one node (i.e., slot multiplexing is not allowed in the static segment). Slot multiplexing (i.e., different nodes owning a slot in different cycles) is allowed in the dynamic segment, and it is up to the application to ensure that the previous characteristics are met.

9.3.2.1.2 Transmit buffer assignment

A transmit buffer shall be assigned to a transmission slot based on the slot identifier of the transmission slot and the identifier of the channel on which the transmission shall occur.

In the static segment possible transmit buffer assignments are

1. assignment to channel A, or
2. assignment to channel B, or
3. assignment to channel A and to channel B.

In the dynamic segment possible transmit buffer assignments are

1. assignment to channel A, or
2. assignment to channel B.

Each transmit buffer shall contain the length *MessageLength* of the message held in the respective transmit buffer.

Multiple transmit buffers may be assigned to the same communication slot on the same channel. In this case it shall be possible to identify a transmit buffer for transmission using the value of the cycle counter to which the specific communication slot relates.

Each transmit buffer shall be associated with a transmit buffer valid flag[106] that denotes whether the message contained in the transmit buffer is valid or not.

For each transmit buffer the CHI shall ensure that the protocol either

1. is provided with a consistent set of valid message data from the transmit buffer, or
2. receives indication that a consistent read of message data is not possible or that the transmit buffer contains no valid message.

[106] The specific mechanism for determining when the information is valid is not specified. For example, the CHI may automatically invalidate the information if it has not been updated by the host within a certain period of time. These features are implementation dependent. It is expected that most implementations will support at least two behaviors - either the message is transmitted exactly once as the result of a buffer update, or a message is transmitted continuously until the host invalidates the buffer.

9.3.2.1.3 Transmit buffer identification for message retrieval

The protocol interacts with the controller host interface by requesting message data at the start of each communication slot according to the media access control processes defined in Chapter 5. The protocol expects to receive data elements as defined in Definition 5-2.

In response to each request the CHI shall perform the following steps:

1. The CHI shall check whether the slot is assigned to the node on the channel *vChannel* by querying the slot assignment list using *vSlotCounter* and *vChannel*.
2. If the specific communication 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 slot assignment list.
4. The transmit buffer shall be identified based on *vSlotCounter*, *vCycleCounter*, and *vChannel*.
5. If no transmit buffer is assigned then the CHI shall signal to the protocol that the communication slot is assigned but without any message data available by setting *vTCHI!TxMessageAvailable* to false and returning *vTCHI*.
6. If a transmit buffer is assigned then a consistent read of its data shall be performed.
7. If a consistent read is not possible, i.e. the transmit buffer is blocked by the host, or the transmit buffer valid flag is set to false, then the CHI shall signal to the protocol that the communication slot is assigned but without any message data available by setting *vTCHI!TxMessageAvailable* to false and returning *vTCHI*.
8. The CHI shall signal to the protocol that the communication slot is assigned with message data available by setting
   a. *vTCHI!TxMessageAvailable* to true,
   b. *vTCHI!PPIndicator* to the value retrieved from the transmit buffer associated control 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*.

9.3.2.1.4 Transmit buffer-related status data

The CHI shall provide message status data for each transmission slot that is assigned to the node on a per channel basis.

For each static and dynamic transmission slot the CHI shall provide the following status data to the host for each assigned channel:

1. a frame transmitted flag that shall be set to true if message data was provided to the protocol engine for transmission and set to false by host access,[107]
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),

[107] Access by host can mean: resetting the frame transmitted flag directly; resetting it as a consequence of access to another flag, e.g. "transmit buffer valid" flag; or by write access to the payload data itself.

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 be set if a valid frame was received in the transmission slot (*vSS!ValidFrame* set to true) or cleared if no valid frame was received in the transmission slot (*vSS!ValidFrame* set to false).

9.3.2.2 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.

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.

9.3.2.2.1 Reception slot subscription and receive buffer assignment

For each communication slot the protocol provides a tuple of values to the CHI consisting of a slot status *vSS*, and the contents of the first semantically valid frame *vRF* that was received in the communication slot, if a semantically valid frame was received in the corresponding communication slot. Different mechanisms within the CHI define how a specific receive buffer is selected based on this tuple.

In general, a receive buffer may be identified using one or more of the following selection criteria:[108]

1. the slot identifier *vSS!SlotCount* of the communication slot to which the tuple relates,
2. the channel identifier *vSS!Channel* of the communication slot to which the tuple relates,
3. the cycle counter *vSS!CycleCount* of the communication slot to which the tuple relates.

For frames received in the static segment, one or more of the following channel assignments may be supported:

1. receive buffer assigned to channel A, or,
2. receive buffer assigned to channel B, or
3. receive buffer assigned to both channel A and channel B. In this case the receive buffer shall contain the first semantically valid frame received on either channel A or on channel B within the corresponding slot.

For frames received in the dynamic segment, one or more of the following channel assignments shall be supported:

1. receive buffer assigned to channel A, or,

[108] An additional optional selection criteria based on message ID filtering is specified in section 9.3.3.3.

2. receive buffer assigned to 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 indication that a consistent read of its data is not possible.

For each receive buffer the CHI shall ensure that the protocol is able to write 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 a flag shall be provided through which the host can assess that receive buffer contents were lost.

Each receive buffer shall hold up to a buffer specific bound number 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*.

9.3.2.2.2 Receive buffer contents

Each receive buffer shall contain slot status-related data as well as frame contents-related data.

9.3.2.2.2.1 Slot status-related data

A snapshot of the following slot status-related protocol variables shall be provided in the CHI for each assigned receive buffer at the end of the respective communication slot:

1. the valid frame flag *vSS!ValidFrame* (in accordance with the corresponding channel-specific FSP process),
2. the syntax error flag *vSS!SyntaxError* (in accordance with the corresponding channel-specific FSP process),
3. the content error flag *vSS!ContentError* (in accordance with the corresponding channel-specific FSP process),
4. the boundary violation flag *vSS!BViolation* (in accordance with the corresponding channel-specific FSP process).

In addition a snapshot of the following status data shall be provided within an assigned receive buffer if a receive buffer is assigned to both channel A and channel B:

1. the enumeration *vSS!Channel* (in accordance with the corresponding channel-specific FSP process) to identify the channel on which the frame was received.

A snapshot of the following slot status-related protocol variables shall be provided in the CHI for each assigned receive buffer at the end of the respective communication slot if a valid frame was received in the respective slot:

1. the flag *vRF!Header!NFIndicator* (in accordance with the corresponding channel-specific FSP process),
2. the flag *vRF!Header!PPIndicator* (in accordance with the corresponding channel-specific FSP process).

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[109] and semantically[110] valid frames received in a static or dynamic slot, respectively.

[109] 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.

[110] A semantically valid frame is a syntactically valid frame that also fulfils a set of content related criteria.

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	-
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 FES of an otherwise valid frame. In the latter case, there may or may not be additional activity.

9.3.2.2.2.2 Frame contents-related data

A snapshot of the following frame contents-related protocol variables shall be provided in the CHI for each assigned receive buffer 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):

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 null frame indicator *vRF!Header!NFIndicator*,

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 assigned receive buffer,
11. *BufferLength* number of two-byte payload data words from *vRF!Payload*, if *vRF!Header!Length* exceeds the buffer length *BufferLength* of the assigned receive buffer.[111]

9.3.3 CHI Services

9.3.3.1 Macrotick timer service

9.3.3.1.1 Absolute timers

The node shall provide at least one absolute timer that may be set to an absolute time in terms of cycle count and macrotick, i.e. the timer is set to expire at a determined macrotick in a determined communication cycle.

It shall be possible to activate an absolute timer as long as the protocol is in either the *POC:normal active* state or the *POC:normal passive* state.

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.

9.3.3.1.2 Relative timers

The node may provide one or more relative timers that may be set to a relative time in terms of macrotick.[112]

It shall be possible to activate a relative timer as long as the protocol is in either the *POC:normal active* state or the *POC:normal passive* state.

All relative 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.

9.3.3.2 Interrupt service

The interrupt service provides a configurable interrupt mechanism to the host based on a set of interrupt sources.

It shall be possible for the host to enable and disable each interrupt individually.

It shall be possible for the host to enable and disable all interrupts without having to enable or disable each interrupt individually.

It shall be possible for the host to clear each interrupt individually.

An interrupt status shall be provided that reflects the status of each interrupt regardless of whether the interrupt is enabled or disabled.

At least one interrupt request shall be available that is raised when a timer has elapsed.

[111] The host can assess such a truncation through the data element *vRF!Header!Length*.

[112] The exact behavior of the relative timer is implementation specific. In particular, the point at which the timer begins to decrement may vary from implementation to implementation. Also note that the duration of a macrotick varies from macrotick to macrotick, especially during the offset correction phase. As a result, the duration of a relative timer can vary substantially depending on when in the cycle it is activated.

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.

Support for the message ID filtering service by a particular device is optional.

Message ID filtering is performed by exchanging a message ID in selected message ID enabled frames within the dynamic segment of the communication cycle that have the payload preamble indicator set to one in the header of the frame.

To support this service minimally the following data needs to be maintained by the CHI:

1. The message buffer control data shall hold the payload preamble indicator for each transmit buffer so that the host can configure whether the message contains a message ID or not. If no message ID service is supported then the CHI shall provide a payload preamble indicator value of zero to the protocol for all transmit buffers that are assigned to the dynamic segment.
2. The message buffer configuration data shall hold a message ID filter for each receive buffer that is involved with message ID filtering.

For each semantically valid frame that was received in the dynamic segment of the communication cycle and that contains a message ID the CHI shall apply this message ID as an additional selection criteria for receive buffer identification.

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 startup and shutdown decisions based on the actual application state.

Support for the network management service by a particular device is optional.

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.

To support this service minimally the following data needs to be maintained by the CHI:

1. The protocol configuration data shall contain the number of bytes *gNetworkManagementVectorLength* contained in the network management vector.
2. The message buffer control data shall hold the payload preamble indicator for each transmit buffer so that the host can configure whether the message contains a network management vector or not. If no network management service is supported then the CHI shall provide a payload preamble indicator value of zero to the protocol for all transmit buffers that are assigned to the static segment.

Throughout each communication cycle the CHI shall maintain an accrued network management vector by applying a bit-wise OR between each network management vector received on each channel (regardless of whether the frame is subscribed to a receive buffer) and the accrued network management vector.

1. The protocol status data shall contain a snapshot of the accrued network management vector that shall be updated at the end of each communication cycle and shall remain available until it is overwritten by the next snapshot at the end of the subsequent communication cycle as long as the protocol is in either the *POC:normal active* state or the *POC:normal passive* state.

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.

Appendix A System Parameters

A.1 Protocol constants

Name	Description	Value
cCASActionPointOffset	Initialization value of the CAS action point offset timer.	1 MT
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 a 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
cdCASRxLowMin	Lower limit of the CAS acceptance window.	29 gdBit
cdCycleMax	Maximum cycle length.	16000 µs
cdFES	Duration of the Frame End Sequence.	2 gdBit
cdFSS	Duration of the Frame Start Sequence.	1 gdBit
cdMaxMTNom[a]	Maximum duration of a nominal macrotick. Each implementation must be able to support macroticks of at least this length. Different implementations may support higher values.	6 µs
cdMinMTNom[b]	Minimum duration of a nominal macrotick. Each implementation must be able to support macroticks of at least this length. Different implementations may support lower values.	1 µs
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

Table A-1: Protocol constants.

Name	Description	Value
cdWakeupSymbolTxLow	Duration of low phase of a transmitted wakeup symbol.	6 µs
cdWakeupSymbolTxIdle	Duration of the idle phase between two low phases inside a wakeup pattern.	18 µs
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
cPayloadLengthMax	Maximum length of the payload segment of a frame.	127 two-byte-words
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
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
cSyncNodeMax	Maximum number of sync nodes in a 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 parameter is only introduced to be able to define a minimum conformance class range that all implementations must support.

[b] This parameter is only introduced to be able to define a minimum conformance class range that all implementations must support.

A.1.1 cdCASRxLowMin

The lower limit of the acceptance window for a collision avoidance symbol (CAS) must meet the following constraint:

Constraint 1:

cdCASRxLowMin[gdBit]= **floor(** *cdCAS*[gdBit] * (1 - *cClockDeviationMax*) / (1 + *cClockDeviationMax*) **)**

= 29 gdBit

As a result, the constant *cdCASRxLowMin* is 29 gdBit.

A.2 Physical layer constants

Name	Description	Value
cPropagationDelayMax	Maximum propagation delay from the falling edge (in the BSS) in the transmit signal of node M to corresponding falling edge at the receiver of node N.	2.5 µs
cdTxMax[a]	Longest possible period of continuous transmission activity for a valid FlexRay configuration.	1433 µs

Table A-2: Physical layer constants.

[a] The value of *cdTxMax* is the lower bound of *dBranchActive* in [EPL05].

Additional parameters related to the Electrical Physical Layer and active/passive stars may be found in [EPL05].

A.2.1 cdTxMax

The value of *cdTxMax* can be determined in the following way:

Constraint 2:

cdTxMax = **max(** *adTxMax* **)**

To calculate the maximum of *adTxMax* the longest possible frames are taken into account (*aFrameLengthStatic*^{Max} = *aFrameLengthDynamic*^{Max}, see B.4.9).

[1] *adTxMax*[µs] >= (*aFrameLengthStatic*[gdBit] + 2 gdBit) * *gdBitMax*[µs/gdBit] + *gdMinislot*[MT] * *gdMacrotick*[µs/MT] * (1 + *cClockDeviationMax*)

Bit Rate [MBit/s]	2.5	5	10
aFrameLengthStatic^{Max}[gdBit]	2628	2631	2638
gdBitMax[µs]	0.4006	0.2003	0.10015
gdMinislot^{Max}[µs]	63		
gdMacrotick^{Max}[µs]	6		
adTxMax[µs]	**1433**	**906**	**643**

Table A-3: Calculation of maximum values for *adTxMax*.

As a result, *cdTxMax* is 1433 µs.

A.3 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-4: 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. The listed parameters will be part of the definition of FlexRay conformance classes.[113] 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.

Currently the only bit rate defined for FlexRay is 10 Mbit/s. There is, however, the intent that the protocol will be expanded in the future to include bit rates lower than 10 Mbit/s. The configuration ranges shown in this appendix reflect bit rates of between 2.5 Mbit/s and 10 Mbit/s. Changes in the range of supported bit rates would also change the required configuration ranges of some parameters. Also, this range should not be interpreted as indicating that all possible bit rates between 2.5 Mbit/s and 10 Mbit/s would be acceptable.

B.2 Global cluster parameters

B.2.1 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
gdActionPointOffset	Number of macroticks the action point is offset from the beginning of a static slot or symbol window.	1 - 63 MT
gdCASRxLowMax	Upper limit of the CAS acceptance window.	67 - 99 gdBit
gdDynamicSlotIdlePhase	Duration of the idle phase within a dynamic slot.	0 - 2 Minislot
gdMinislot	Duration of a minislot.	2 - 63 MT
gdMinislotActionPoint-Offset	Number of macroticks the minislot action point is offset from the beginning of a minislot.	1 - 31 MT
gdStaticSlot	Duration of a static slot.[a]	4 - 661 MT
gdSymbolWindow	Duration of the symbol window.	0 - 142 MT
gdTSSTransmitter	Number of bits in the Transmission Start Sequence.	3 - 15 gdBit

Table B-1: Global protocol relevant parameters.

[113] Exceptions for parameters which should not be part of a conformance class are indicated by footnote.

Name	Description	Range
gdWakeupSymbolRxIdle	Number of bits used by the node to test the duration of the 'idle' portion of a received wakeup symbol. Duration is equal to (*gdWakeupSymbolTxIdle* - *gdWakeupSymbolTxLow*)/2 minus a safe part. (Collisions, clock differences, and other effects can deform the Tx-wakeup pattern.)	14 - 59 gdBit
gdWakeupSymbolRxLow	Number of bits used by the node to test the LOW portion of a received wakeup symbol. This lower limit of zero bits has to be received to detect the LOW portion by the receiver. The duration is equal to *gdWakeupSymbolTxLow* minus a safe part. (Active stars, clock differences, and other effects can deform the Tx-wakeup pattern.)	11 - 59 gdBit
gdWakeupSymbolRxWindow	The size of the window used to detect wakeups. Detection of a wakeup requires a low and idle period (from one WUS) and a low period (from another WUS) to be detected entirely within a window of this size. The duration is equal to *gdWakeupSymbolTxIdle* + 2 * *gdWakeupSymbolTxLow* plus a safe part. (Clock differences and other effects can deform the Tx-wakeup pattern.)	76 - 301 gdBit
gdWakeupSymbolTxIdle	Number of bits used by the node to transmit the 'idle' part of a wakeup symbol. The duration is equal to *cdWakeupSymbolTxIdle*.	45 - 180 gdBit
gdWakeupSymbolTxLow	Number of bits used by the node to transmit the LOW part of a wakeup symbol. The duration is equal to *cdWakeupSymbolTxLow*.	15 - 60 gdBit
gListenNoise	Upper limit for the startup listen timeout and wakeup listen timeout in the presence of noise. It is used as a multiplier of the node parameter *pdListenTimeout*.	2 - 16
gMacroPerCycle	Number of macroticks in a communication cycle.	10 - 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.	*gMaxWithoutClockCorrectionPassive* - 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. Note that *gMaxWithoutClockCorrectionPassive* <= *gMaxWithoutClockCorrectionFatal* <= 15.	1 - 15 even/odd cycle pairs
gNumberOfMinislots	Number of minislots in the dynamic segment.	0 - 7986

Table B-1: Global protocol relevant parameters.

Name	Description	Range
gNumberOfStaticSlots	Number of static slots in the static segment.	2 - *cStaticSlotIDMax*
gOffsetCorrectionStart	Start of the offset correction phase within the NIT, expressed as the number of macroticks from the start of cycle.	9 - 15999 MT
gPayloadLengthStatic	Payload length of a static frame.[b]	0 - *cPayloadLengthMax* two-byte-words
gSyncNodeMax	Maximum number of nodes that may send frames with the sync frame indicator bit set to one.	2 - *cSyncNodeMax*

Table B-1: Global protocol relevant parameters.

[a] This parameter is also used in the FlexRay Electrical Physical Layer Specification [EPL05].

[b] All static frames in a cluster have the same payload length.

B.2.2 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
gAssumedPrecision	Assumed precision of the application network.	0.15 - 11.7 µs
gChannels	The channels that are used by the cluster.	[A, B, A&B]
gClusterDriftDamping	The cluster drift damping factor, based on the longest microtick *gdMaxMicrotick* used in the cluster. Used to compute the local cluster drift damping factor *pClusterDriftDamping*.	0 - 5 µT
gdBit	Nominal bit time (see formula [7]).	*cSamplesPerBit* * *gdSampleClockPeriod* [µs]
gdBitMax[a]	Maximum bit time taking into account the allowable clock deviation of each node.	*gdBit* * (1 + *cClockDeviationMax*) [µs]
gdBitMin[b]	Minimum bit time taking into account the allowable clock deviation of each node.	*gdBit* * (1 - *cClockDeviationMax*) [µs]
gdCycle	Length of the cycle, expressed in µs.	10 µs[c] - *cdCycleMax*
gdMacrotick	Duration of the cluster wide nominal macrotick, expressed in µs (see formula [8]).	1 - 6 µs
gdMaxInitializationError	Maximum timing error that a node may have following integration.	0 - 11.7 µs

Table B-2: Global protocol related parameters.

Name	Description	Range
gdMaxMicrotick[d]	Maximum microtick length of all microticks configured within a cluster (see formula [2]).	*pdMicrotick* [µs]
gdMaxPropagation-Delay	Maximum propagation delay of a cluster (see Constraint 3).	<= *cPropagation-DelayMax* [µs]
gdMinPropagation-Delay	Minimum propagation delay of a cluster.	<= *gdMaxPropagationDelay* [µs]
gdNIT	Duration of the Network Idle Time.	2 - 805 MT
gdSampleClockPeriod	Sample clock period.	[0.0125, 0.025, 0.05 µs]
gNetworkManagementVectorLength	Length of the Network Management vector in a cluster.	0 - 12 bytes
gOffsetCorrectionMax	Cluster global magnitude of the maximum necessary offset correction value.	0.15 - 383.567 µs

Table B-2: Global protocol related parameters.

[a] This parameter is only introduced for configuration constraints.

[b] This parameter is only introduced for configuration constraints.

[c] See the minimum constraint for *gMacroPerCycle*. Maximum value is given by *cdCycleMax*.

[d] This parameter is only introduced for configuration constraints.

B.2.3 Physical layer relevant

For detailed information refer to [EPL05].

B.3 Node parameters

B.3.1 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 flag that controls the transition to the *POC:halt* state due to a 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

Table B-3: Local node protocol relevant parameters.

Name	Description	Range
pChannels	Channels to which the node is connected.	[A, B, A&B]
pdAcceptedStartupRange	Expanded range of measured clock deviation allowed for startup frames during integration.	0 - 1875 µT
pClusterDriftDamping	Local cluster drift damping factor used for rate correction.	0 - 20 µT
pDecodingCorrection	Value used by the receiver to calculate the difference between primary time reference point and secondary time reference point.	14 - 143 µT
pDelayCompensation[A], *pDelayCompensation[B]*	Value used to compensate for reception delays on the indicated channel. This covers assumed propagation delay up to *cPropagationDelayMax* for microticks in the range of 0.0125 µs to 0.05 µs. In practice, the minimum of the propagation delays of all sync nodes should be applied.	0 - 200 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.	1284 - 1283846 µT
pdMaxDrift	Maximum drift offset between two nodes that operate with unsynchronized clocks over one communication cycle.	2 - 1923 µT
pExternOffsetCorrection	Number of microticks added or subtracted to the NIT to carry out a host-requested external offset correction.	0 - 7 µT
pExternRateCorrection	Number of microticks added or subtracted to the cycle to carry out a host-requested external rate correction.	0 - 7 µT
pKeySlotId	ID of the slot used to transmit the startup frame, sync frame, or designated single slot frame.	1 - *cStaticSlotIdMax*
pKeySlotUsedForStartup	Flag indicating whether the Key Slot is used to transmit a startup frame. If *pKeySlotUsedForStartup* is set to true then *pKeySlotUsedForSync* must also be set to true.	Boolean
pKeySlotUsedForSync	Flag indicating whether the Key Slot is used to transmit a sync frame. If *pKeySlotUsedForStartup* is set to true then *pKeySlotUsedForSync* 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 - 7980 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

Table B-3: Local node protocol relevant parameters.

Name	Description	Range
pMicroInitialOffset[A], *pMicroInitialOffset[B]*	Number of microticks between the closest macrotick boundary described by *pMacroInitialOffset[Ch]* and 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.	640 - 640000 µT
pOffsetCorrectionOut	Magnitude of the maximum permissible offset correction value.	13 - 15567 µT
pRateCorrectionOut	Magnitude of the maximum permissible rate correction value.	2 - 1923 µT
pSingleSlotEnabled	Flag indicating whether or not the node shall enter single slot mode following startup.	Boolean
pWakeupChannel	Channel used by the node to send a wakeup pattern.	[A, B]
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.	2 - 63

Table B-3: Local node protocol relevant parameters.

B.3.2 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[a]	Duration of a microtick.	*pSamplesPerMicrotick* * *gdSampleClockPeriod* [µs]
pMicroPerMacroNom	Number of microticks per nominal macrotick that all implementations must support.	*cMicroPerMacroNomMin* - *cMicroPerMacroNomMax*
pPayloadLengthDynMax	Maximum payload length for dynamic frames.	0 - *cPayloadLengthMax*
pSamplesPerMicrotick[b]	Number of samples per microtick.	[1, 2, 4]

Table B-4: Local node protocol related parameters.

[a] Shall not be part of the conformance test due to implementation dependency.

[b] Shall not be part of the conformance test due to implementation dependency.

B.3.3 Physical layer relevant

For values of the following parameter please refer to [EPL05].

Name	Description
dStarTruncation	Interval by which a transmitted TSS is shortened by one 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.
dBDRxia	Activity reaction time. Time by which a transmission becomes shortened in a receiving node (when bus is switched from idle to active).
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 active driven bit was HIGH the idle detection in the CC is not delayed.
dBDTxia	Activity reaction time. Time by which a transmission becomes shortened in a transmitting node (when bus is switched from idle to active).
dBDTxai	Idle reaction time. Time by which a transmission becomes lengthened in a transmitting node (when bus is switched from active to idle).
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.
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.

Table B-5: Local node physical layer related parameters.

B.4 Calculation of configuration parameters

Following abbreviations and functions are used in this section:

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(**x;y**)** returns the maximum value from a set of values {x;y}.
4. Function **min(**x;y**)** returns the minimum value from a set of values {x;y}.
5. Function **round(**x**)** returns the rounded integer value of x.
6. Function **if(**c;x;y**)** returns x if condition c is true, otherwise y.
7. [ ] denotes units.

B.4.1 Attainable precision

Various error terms influence the attainable precision of the FlexRay clock synchronization algorithm (see [Mül01] and [Ung02]). In order to choose proper configuration parameters it is necessary to know the attainable precision of the application network.

B.4.1.1 Propagation Delay

A parameter for the maximum propagation delay *gdMaxPropagationDelay*[µs] of the cluster is introduced with

Constraint 3:

gdMaxPropagationDelay[Ch][µs] = **max(** $pdBDTx[Ch]_M$[µs] + $LineLength[Ch]_{M,N}$[m] * T_0[µs/m] +

$$\sum_{k \in \{nStar[Ch]_{MN}\}} pdStarDelay_k \quad [\mu s] + pdBDRx[Ch]_N[\mu s] \;)_{M,N}$$

gdMaxPropagationDelay[µs] = **max(** *gdMaxPropagationDelay[A]*[µs], *gdMaxPropagationDelay[B]*[µs] **)**

<= *cPropagationDelayMax*

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.

- $pdBDTx[Ch]_M$ is the maximum propagation delay of the transmitter of node M on Channel *Ch* and represents the maximum of the propagation delay of a positive edge (*dBDTx01*) and a negative edge (*dBDTx10*) inside the transmitter
- $pdBDRx[Ch]_N$ is the maximum propagation delay of the receiver of node N on Channel *Ch* and represents the maximum of the propagation delay of a positive edge (*dBDRx01*) and a negative edge (*dBDRx10*) inside the receiver.
- $pdStarDelay_k$ is the propagation delay of star *k*.
- $LineLength[Ch]_{M,N}$ is the real line length between node M and node N on Channel *Ch*.
- {$nStar[Ch]_{MN}$} is the number of active stars between node M and N on Channel *Ch*.
- The propagation delay per unit length of a line T_0 is approximately 0.01µs/m.

The value of *gdMaxPropagationDelay* should be estimated for the given network topology using the estimated propagation delays of line, star and transmitter. If this is not possible or a worst case estimation is needed, a value from Table B-6 should be chosen. The $gdMaxPropagationDelay^{Max}$ parameter from Table B-6 is also used to derive additional constraints on other parameters.

In some cases the propagation delay of a star may be up to 0.7µs (if, for example, the star includes a signal refresh feature). It is acceptable to use a star with this increased propagation delay if the overall propagation delay *gdMaxPropagationDelay* is always less than or equal to the upper limit of *cPropagationDelayMax*.

nStar	**0**	**1**	**2**
LineLength[m]	24	48	72
$pdBDTx^{Max}$[µs]	0.1		
$pdBDRx^{Max}$[µs]	0.1		
$pdStarDelay^{Max}$[µs]	0.25		
$gdMaxPropagationDelay^{Max}$[µs]	**0.44**	**0.93**	**1.42**

Table B-6: Maximum propagation delay in a cluster. Maximum values for *LineLength*, *pdBDTx*, *pdBDRx* and *pdStarDelay* were chosen according to limits described in [EPL05].

B.4.1.2 Worst-case precision

First of all, a parameter defining the maximum used microtick of a cluster is introduced:

[2] *gdMaxMicrotick*[µs] = **max(** { x | x = *pdMicrotick* of each node } **)**

The worst-case error before the correction is given by

[3] *aWorstCasePrecision*[µs] = (34 µT + 20 * *gClusterDriftDamping*[µT]) * *gdMaxMicrotick*[µs/µT] + 2 * *gdMaxPropagationDelay*[µs]

It is important to note that the attainable precision directly depends on network topology (*gdMaxPropagationDelay*) and the maximum microtick used in the cluster (*gdMaxMicrotick*).

As explained in [Mül01] and [Ung02], it can be proven 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.

***gdMaxMicrotick*[µs]**	**0.100**	**0.050**	**0.025**	**0.0125**
gClusterDriftDamping^{Max}[µT]	5			
gdMaxPropagationDelay[µs] = *cPropagationDelayMax*[µs]	2.5			
aWorstCasePrecision^{Max}[µs]	**18.4**	**11.7**	**8.35**	**6.675**

Table B-7: Calculation of the worst-case precision.

For further parameter calculations it is assumed that *gdMaxMicrotick*[µs] = 0.05 µs or a smaller value. Therefore the worst-case precision is assumed to be 11.7 µs.

B.4.1.3 Best-case precision

The best-case precision can be calculated using a simplified equation [3] which does not take Byzantine errors into account.

[4] *aBestCasePrecision*[µs] >= (12 µT + 6 * *gClusterDriftDamping*[µT]) * *gdMaxMicrotick*[µs/µT] + *gdMaxPropagationDelay*[µs] - *gdMinPropagationDelay*[µs]

It is not possible to reach a precision better than calculated in [4].

***gdMaxMicrotick*[µs]**	**0.100**	**0.050**	**0.025**	**0.0125**
gClusterDriftDamping^{Min}[µT]	0			
min(*gdMaxPropagationDelay*[µs] - *gdMinPropagationDelay*[µs] **)**	0			
aBestCasePrecision^{Min}[µs]	**1.2**	**0.6**	**0.3**	**0.15**

Table B-8: Calculation of the best-case precision.

B.4.1.4 Assumed precision

An assumed precision of the cluster, *gAssumedPrecision*[µs] is introduced. This parameter depends on the parameters *gdMaxPropagationDelay*, *gdMaxMicrotick* and *gClusterDriftDamping*. The *gAssumedPrecision* parameter is necessary to derive additional constraints on other timing parameters. This parameter is given by

Constraint 4:

aBestCasePrecision[µs] <= *gAssumedPrecision*[µs] <= *aWorstCasePrecision*[µs]

For purposes of configuration of the cluster, it is suggested that *gAssumedPrecision* should be:

[5] *gAssumedPrecision*[µs] = *aWorstCasePrecision*[µs] = (34 µT + 20 * *gClusterDriftDamping*[µT]) * *gdMaxMicrotick*[µs/µT] +2 * *gdMaxPropagationDelay*[µs]

B.4.2 Definition of microtick and macrotick

The parameter *pdMicrotick* is usually not a configuration parameter of an implementation. It is introduced here because it is used to derive various parameter constraints.

The microtick length (*pdMicrotick*) is implementation 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.

[6] *pdMicrotick* = *pSamplesPerMicrotick* * *gdSampleClockPeriod*

[7] *gdBit* = *cSamplesPerBit* * *gdSampleClockPeriod*

Table B-9 shows the possible microtick lengths depending on *pSamplesPerMicrotick* and *gdSampleClock-Period*. Table B-10 shows the possible nominal macrotick lengths (*gdMacrotick*) on a range of the microticks per macrotick parameter (*pMicroPerMacroNom*) of between 40 to 240. In practice, it is assumed that the typical microtick range is *pdMicrotick* = 0.0125 µs ... 0.05 µs because of this parameter's direct influence to the attainable precision.

***gdSample-ClockPeriod*[µs]**	***pSamplesPerMicrotick***		
	1	2	4
0.0125	0.0125	0.025	0.050
0.0250	0.025	0.050	0.100
0.0500	0.050	0.100	-

Table B-9: *pdMicrotick*[µs] depending on *pSamplesPerMicrotick* and *gdSampleClockPeriod*.

pMicroPer-MacroNom	***pdMicrotick*[µs]**			
	0.0125	0.025	0.050	0.100
40	-	1	2	4
60	-	1.5	3	6
80	1	2	4	-
120	1.5	3	6	-
240	3	6	-	-

Table B-10: *gdMacrotick*[µs] depending on *pMicroPerMacroNom* and *pdMicrotick*.

Some parameter constraints depend on the microtick length of the node and the nominal macrotick length of the cluster. To define a minimum parameter range for these parameters, that is the basis of conformance tests, a typical microtick length of 0.025 µs and the resulting nominal macrotick range of 1 µs to 6 µs (Constraint 5) are assumed (green fields within Table B-9 and B-10).

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 5:

cdMinMTNom[µs] <= *gdMacrotick*[µs] <= *cdMaxMTNom*[µs]

Depending on the desired macrotick length the following additional constraint must be met.

Constraint 6:

gdMacrotick[µs] >= *cMicroPerMacroNomMin*[µT] * *pdMicrotick*[µs/µT]

The macrotick, expressed in microticks, is given by

[8] *pMicroPerMacroNom*[µT/MT]= *gdMacrotick*[µs/MT] / *pdMicrotick*[µs/µT]

= *pMicroPerCycle*[µT] / *gMacroPerCycle*[MT]

Note that *pMicroPerMacroNom* may have a fractional part (i.e., it does not need to be an integral number of microticks).

Note that *pMicroPerCycle* might have rounding errors according to section 8.2.3.

For *gdMacrotick* exists an additional constraint because of the startup procedure and *pMicroInitialOffset*:

Constraint 7:

gdMacrotick[µs] <= *aFrameLengthStatic*[gdBit] * *gdBitMin*[µs/gdBit]

However, the fulfillment of this constraint is given in any case because of the minimum frame length of 80 bits, a bit duration of around 100 ns and a maximum macrotick length of 6 µs.

B.4.3 gdMaxInitializationError

Consider the following assumptions:

- The maximum initialization error that an integrating node may have is

  0 µs <= *gdMaxInitializationError*[µs] <= *gAssumedPrecision*[µs]

Constraint 8:

gAssumedPrecision[µs] >= *gdMaxInitializationError*[µs] >=

2 * (*gdMaxMicrotick*[µs] * (1 + *cClockDeviationMax*)) + *gdMaxPropagationDelay*[µs]

The upper limit is given by

gdMaxInitializationError[µs] = *gAssumedPrecision*[µs] = *aWorstCasePrecision*[µs] = 11.7µs.

B.4.4 pdAcceptedStartupRange

Consider the following assumptions:

- During integration a clock synchronization error greater than the assumed precision may occur and be acceptable.

Constraint 9:

pdAcceptedStartupRange[µT] >= **ceil(** (*gAssumedPrecision*[µs] + *gdMaxInitializationError*[µs]) /

(*pdMicrotick*[µs/µT] * (1 - *cClockDeviationMax*)) **)**

The maximum value of this expression occurs when

- *gAssumedPrecision* = *aWorstCasePrecision*,

- *gdMaxInitializationError* = *aWorstCasePrecision*,
- *pdMicrotick* = 0.0125 µs

and is

pdAcceptedStartupRange = **ceil(** 1874.8 **)** µT = 1875 µT.

B.4.5 pClusterDriftDamping

Consider the following assumptions:

- The drift damping factor *gClusterDriftDamping* is defined in multiples of the longest microtick *gdMaxMicrotick* within the cluster:

  gClusterDriftDamping[µT] = n * 1 µT with n = 0, 1, 2, ..., 5.
- The maximum microtick of a cluster *gdMaxMicrotick* is a multiple m of each nodes local microtick *pdMicrotick*.

  gdMaxMicrotick = m * *pdMicrotick* with m = 1, 2, 4 (see *pSamplesPerMicrotick*).
- The local drift damping *pClusterDriftDamping* is calculated by

Constraint 10:

pClusterDriftDamping[µT] <= *gdMaxMicrotick*[µs] / *pdMicrotick*[µs] * *gClusterDriftDamping*[µT]

Constraint 10 should be treated as a recommendation. In practice it is expected that the drift damping factor is chosen such that

pClusterDriftDamping ~ *gClusterDriftDamping*.

The upper limit of *pClusterDriftDamping* is given by *gdMaxMicrotick*/*pdMicrotick* = 4. Therefore

pClusterDriftDamping = 4 * 5 µT = 20 µT.

B.4.6 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 *gdMinPropagationDelay*[µs].

Constraint 11:

gdActionPointOffset[MT] >= **ceil(** (*gAssumedPrecision*[µs] - *gdMinPropagationDelay*[µs]) /

(*gdMacrotick*[µs/MT] * (1 - *cClockDeviationMax*)) **)**

In practice the minimum propagation delay *gdMinPropagationDelay* may be set to zero.

In order to prevent the possibility of the creation of cliques[114] during startup an additional safety margin must be added. In this case Constraint 12 replaces Constraint 11.

Constraint 12:

gdActionPointOffset[MT] >= **ceil(** (2 * *gAssumedPrecision*[µs] - *gdMinPropagationDelay*[µs] +

2 * *gdMaxInitializationError*[µs]) / (*gdMacrotick*[µs/MT] * (1 - *cClockDeviationMax*)) **)**

[114] Clique formation may be possible if more than two coldstart nodes are configured in the cluster under specific error scenarios.

Parameter	Constraint 11	Constraint 12
gAssumedPrecision^{Max}[µs]	11.7	11.7
gdMinPropagationDelay^{Min}[µs]	0	0
gdMaxInitializationError^{Max}[µs]	-	11.7
gdMacrotick^{Min}[µs]	1	1
gdActionPointOffset^{Max}[MT]	**12**	**47**

Table B-11: Calculation of values for *gdActionPointOffset*.

In order to determine the configuration range of the *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.[115] 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.4.7 gdMinislotActionPointOffset

Consider the following assumptions:

- The action point offset should be greater than the attainable precision.

Constraint 13:

gdMinislotActionPointOffset[MT] >= **ceil(** (*gAssumedPrecision*[µs] - *gdMinPropagationDelay*[µs]) / (*gdMacrotick*[µs/MT] * (1 - *cClockDeviationMax*)) **)**

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. *gdMinislotActionPointOffset* can also be different from *gdActionPointOffset* if the worst case precision (causing *gdActionPointOffset*) can be replaced by an assumed "average precision" for the dynamic segment. In both cases, the independent choice of *gdMinislotActionPointOffset* allows increased throughput respective bandwidth within the dynamic segment.

B.4.8 gdMinislot

Consider the following assumptions:

- The minislot action point *gdMinislotActionPointOffset* is greater than or equal to the assumed precision *gAssumedPrecision*.
- The maximum propagation delay *gdMaxPropagationDelay* of the network is taken into account.

Constraint 14:

gdMinislot[MT] >= *gdMinislotActionPointOffset*[MT] + **ceil(** (*gdMaxPropagationDelay*[µs] + *gAssumedPrecision*[µs]) / (*gdMacrotick*[µs/MT] * (1 - *cClockDeviationMax*)) **)**

With the assumptions for *gdMinislotActionPointOffset* given in section B.4.7 and allowing for a margin of safety, the parameter *gdMinislot* must be configurable over a range of 2 to 63 MT.

[115] Note that the *gdActionPointOffset* parameter also specifies the action point in the symbol window.

B.4.9 gdStaticSlot

Consider the following assumptions:

- A frame must consists of at least *gdTSSTransmitter*, *cdFSS*, 80 gdBit header and trailer (with *gPayloadLengthStatic* = 0), and *cdFES*.
- Each payload data word is equal to 2 * (8 gdBit + *cdBSS*[gdBit]) = 20 gdBit.

The length of a frame is given by:

[9] *aFrameLength*[gdBit] = *gdTSSTransmitter*[gdBit] + *cdFSS*[gdBit] + 80 gdBit + *aPayloadLength*[2-byte-word] * 20 gdBit + *cdFES*[gdBit]

Substituting the length of a static frame for *aPayloadLength* results in:

[10] *aFrameLengthStatic* = *aFrameLength* with *aPayloadLength* = *gPayloadLengthStatic*

Additional to the frame length the following

- The maximum length of a frame is increased depending on the clock quality (*cClockDeviationMax*) and the minimum duration of a macrotick.
- The influence of the precision is taken into account by *gdActionPointOffset*.
- An idle detection time of *cChannelIdleDelimiter* between two consecutive frames is considered.
- The maximum propagation delay *gdMaxPropagationDelay* of the cluster must also be taken into account.

Using equation [9] and [10] the minimum static slot length for systems can be calculated by Constraint 15:

Constraint 15:

gdStaticSlot[MT] = 2 * *gdActionPointOffset*[MT] + **ceil(** ((*aFrameLengthStatic*[gdBit] + *cChannelIdleDelimiter*[gdBit]) * *gdBitMax*[µs/gdBit] + *gdMinPropagationDelay*[µs] + *gdMaxPropagationDelay*[µs]) / (*gdMacrotick*[µs/MT] * (1 - *cClockDeviationMax*)) **)**

Bit Rate [MBit/s]	2.5	5	10
gdActionPointOffset^{Min}[MT]	1		
gdActionPointOffset^{Max}[MT]	63		
gdTSSTransmitter ^{Min}[gdBit]	3	4	6
gdTSSTransmitter^{Max}[gdBit]	5	8	15
aFrameLengthStatic^{Min}[gdBit]	86	87	89
aFrameLengthStatic^{Max}[gdBit]	2628	2631	2638
gdBitMax[µs]	0.4006	0.2003	0.10015
gdMinPropagationDelay^{Min}[µs]	0		
gdMinPropagationDelay^{Max}[µs]	2.5		
gdMaxPropagationDelay^{Min}[µs]	0		
gdMaxPropagationDelay^{Max}[µs]	2.5		

Table B-12: Calculation of minimum and maximum values for *gdStaticSlot*.

Bit Rate [MBit/s]	2.5	5	10
gdMacrotick^{Min}[µs]	2	1	
gdMacrotick^{Max}[µs]	6		
gdStaticSlot^{Min}[MT]	**9**	**6**	**4**
gdStaticSlot^{Max}[MT]	**658**	**661**	**397**

Table B-12: Calculation of minimum and maximum values for *gdStaticSlot*.

As a result, the parameter *gdStaticSlot* must be configurable over a range of 4 to 661 MT.

B.4.10 gdSymbolWindow

Consider the following assumptions:

- The length of a symbol (CAS or MTS symbol) is defined by *cdCAS*[116].
- A CAS or MTS symbol has to be sent with a leading transmit start sequence. The symbol window takes this into account by using *gdTSSTransmitter* + *cdCAS*.
- The transmission of a CAS or MTS symbol ends with a LOW bit. Therefore the idle detection in the receiving CC's is delayed by *dBDRxai*.
- The influence of the precision is taken into account by *gdActionPointOffset*.
- After completion of the transmission of the symbol an idle detection time of *cChannelIdleDelimiter* is required.
- The maximum time required to transmit the symbol is increased depending on the clock quality (*cClockDeviationMax*).
- The duration of a macrotick may also be decreased depending on the clock quality.

Constraint 16 describes the calculation of the minimum time for the symbol window

Constraint 16:

gdSymbolWindow[MT] = 2 * *gdActionPointOffset*[MT] + **ceil(** ((*gdTSSTransmitter*[gdBit] + *cdCAS*[gdBit] + *cChannelIdleDelimiter*[gdBit]) * *gdBitMax*[µs/gdBit] + *dBDRxai*[µs] + *gdMinPropagationDelay*[µs] + *gdMaxPropagationDelay*[µs]) / (*gdMacrotick*[µs/MT] * (1 - *cClockDeviationMax*)) **)**

Bit Rate [MBit/s]	2.5	5	10
gdActionPointOffset^{Max}[MT]	63		
gdTSSTransmitter^{Max}[gdBit]	5	8	15
gdBitMax[µs]	0.4006	0.2003	0.10015
dBDRxai^{Max}[µs]	0.4		
gdMinPropagationDelay^{Max}[µs]	2.5		

Table B-13: Calculation of values of *gdSymbolWindow*.

[116] The collision avoidance symbol, CAS, is also used to be the media access test symbol, MTS. It is the only possible symbol within the symbol window.

Bit Rate [MBit/s]	2.5	5	10
gdMaxPropagationDelay^{Max}[µs]	2.5		
gdMacrotick^{Min}[µs]	2	1	
gdSymbolWindow^{Min}[MT]	**0**[a]		
gdSymbolWindow^{Max}[MT]	**138**	**142**	**138**

Table B-13: Calculation of values of *gdSymbolWindow*.

[a] 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 142 MT.

B.4.11 gMacroPerCycle

The number of macroticks per cycle is based on the cycle duration and the macrotick length.

Constraint 17:

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	1
gMacroPerCycle^{Max}[MT]	**8000**	**16000**	**16000**

Table B-14: Calculation of the maximum value of *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 18:

gMacroPerCycle[MT] = *gdStaticSlot*[MT] * *gNumberOfStaticSlots* + *adActionPointDifference*[MT] +
gdMinislot[MT] * *gNumberOfMinislots* + *gdSymbolWindow*[MT] + *gdNIT*[MT]

adActionPointDifference[MT] is introduced in B.4.14 and calculated in [12].

Bit Rate [MBit/s]	2.5	5	10
gdStaticSlot^{Min}[MT]	9	6	4
gNumberOfStaticSlots^{Min}	2		
gdMinislot^{Min}[MT]	0		
gNumberOfMinislots^{Min}[Minislot]	0		

Table B-15: Calculation of the minimum value of *gMacroPerCycle*.

Bit Rate [MBit/s]	2.5	5	10
gdSymbolWindow^{Min}[MT]	0		
gdNIT^{Min}[MT]	2		
gMacroPerCycle^{Min}[MT]	**20**	**14**	**10**

Table B-15: Calculation of the minimum value of *gMacroPerCycle*.

As a result, the parameter *gMacroPerCycle* must be configurable over a range of 10 to 16000 MT.

B.4.12 pMicroPerCycle

The cycle length in microticks is implementation dependent and may be calculated using the following equations:

Constraint 19:

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:

- Minimum number of macroticks per cycle to run a cluster with two static slots (see Table B-15:).
- Minimum number of microticks per macrotick.

To calculate the lower boundary a different equation is used:

[11] *pMicroPerCycle*^{Min}[µT]= (2 * *gdStaticSlot*[MT] + *gdNIT*[MT]) * *pMicroPerMacroNom*[µT/MT]
= (4 * *gdActionPointOffset*[MT] + *gdNIT*[MT] + 2 * **ceil(** ((*aFrameLengthStatic*[gdBit] + *cChannelIdleDelimiter*[gdBit]) * *gdBitMax*[µs/gdBit] + *gdMaxPropagationDelay*[µs]) / (*pdMicrotick*[µs/µT] * *pMicroPerMacroNom*[µT/MT]) **)**) ***** *pMicroPerMacroNom*[µT/MT]

Bit Rate [MBit/s]	2.5	5	10
gdActionPointOffset^{Min}[MT]	1		
gdNIT^{Min}[MT]	2		
pMicroPerMacroNom^{Min}[µT][a]	40	40	41
aFrameLengthStatic^{Min}[gdBit] + *cChannelIdleDelimiter*[gdBit]	97	98	100
gdBitMax[µs]	0.4006	0.2003	0.10015
gdMaxPropagationDelay^{Min}[µs]	0		
pdMicrotick^{Max}[µs]	0.1[b]	0.1	0.05
pMicroPerCycle^{Min}[µT]	**1040**	**640**	**656**

Table B-16: Calculation of the minimum of *pMicroPerCycle*.

[a] *pMicroPerMacroNom*[µT] is chosen in a way that *pMicroPerCycle*[µT] results in a minimum.

[b] With *pMicroPerMacroNom*^{Min} = 40 µT and *pdMicrotick* = 0.2 µs the maximum length of *gdMacrotick* would exceed the maximum macrotick length of 6 µs.

Bit Rate [MBit/s]	2.5	5	10
gdCycle^{Max}[µs] = *cdCycleMax*	16000		
pdMicrotick[µs]	0.050 0.100 -	0.025 0.050 0.100	0.0125 0.025 0.050
pMicroPerCycle^{Max}[µT] with *pdMicrotick* = 0.100 µs	**160000**	160000	-
pMicroPerCycle^{Max}[µT] with *pdMicrotick* = 0.050 µs	320000	**320000**	320000
pMicroPerCycle^{Max}[µT] with *pdMicrotick* = 0.025 µs	-	640000	**640000**
pMicroPerCycle^{Max}[µT] with *pdMicrotick* = 0.0125 µs	-	-	1280000

Table B-17: Calculation of the maximum of *pMicroPerCycle*.

The maximum number of microticks per cycle depends on the microtick length. It is not required that any implementation supports any microtick length but for each supported bit rate at least one microtick length and the related number of microticks per cycle must be supported (see Table B-16). To support the minimum macrotick length of 1 µs the bold marked numbers of microtick per cycle should be supported.

As a result, the parameter *pMicroPerCycle* must be configurable at least over a range of 640 to 640000 µT.

B.4.13 gdDynamicSlotIdlePhase

Consider the following assumptions:

- The duration of *gdDynamicSlotIdlePhase*[Minislots] must be greater than or equal to the idle detection time.
- The idle detection time must be calculated based on the uncorrected bit time and therefore equals *cChannelIdleDelimiter* * *gdBitMax*.
- The macroticks may also be shortened by the clock deviation.

Constraint 20:

gdDynamicSlotIdlePhase[Minislot] >= **ceil(** (**ceil(** (*cChannelIdleDelimiter* * *gdBitMax*[µs] + *gAssumedPrecision*[µs] + *gdMaxPropagationDelay*[µs]) / (*gdMacrotick*[µs/MT] * (1 - *cClockDeviationMax*)) **)** - (*gdMinislot*[MT] - *gdMinislotActionPointOffset*[MT])) / *gdMinislot*[MT/Minislot] **)**

Note, to calculate the minimum and the maximum value of *gdDynamicSlotIdlePhase* the Constraint 14 must also be fulfilled.

Bit Rate [MBit/s]	2.5	5	10
gdBitMax[µs]	0.4006	0.2003	0.10015

Table B-18: Calculation of the minimum and maximum of *gdDynamicSlotIdlePhase*.

Bit Rate [MBit/s]	2.5	5	10
gAssumedPrecision^{Min}[µs]	0.5		
gdMaxPropagationDelay^{Min}[µs]	0		
gdMaxPropagationDelay^{Max}[µs]	2.5		
gdMacrotick^{Min}[µs]	2	1	1
gdMacrotick^{Max}[µs]	6		
gdMinislot^{Min}[MT]	2		
gdMinislot^{Max}[MT]	63		
gdDynamicSlotIdlePhase^{Min}[Minislot]	**0**	**0**	**0**
gdDynamicSlotIdlePhase^{Max}[Minislot]	**2**	**2**	**1**

Table B-18: Calculation of the minimum and maximum of *gdDynamicSlotIdlePhase*.

The parameter *gdDynamicSlotIdlePhase* must be configurable over a range of 0 to 2 Minislots.

B.4.14 gNumberOfMinislots

Consider the following:

[12] *adActionPointDifference*[MT] = 0 **if** *gdActionPointOffset* <= *gdMinislotActionPointOffset*

= 0 **if** *gNumberOfMinislots* = 0

= *gdActionPointOffset* - *gdMinislotActionPointOffset* **else**

Constraint 21:

gNumberOfMinislots[Minislot] = (*gMacroPerCycle*[MT] - *gdNIT*[MT] - *adActionPointDifference*[MT] -

gNumberOfStaticSlots * *gdStaticSlot*[MT] - *gdSymbolWindow*[MT]) / *gdMinislot*[MT/Minislot]

gNumberOfMinislots is always an integer. To fulfill Constraint 21 the parameters on the right side of the equation must be chosen so that *gNumberOfMinislots* results in an integer.[117]

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	22	-	-
gdStaticSlot^{Min}[MT] with *gdMacrotick* = 1 µs	-	22	13
gdSymbolWindow^{Min}[MT]	0		

Table B-19: Calculation of maximum values for *gNumberOfMinislots*.

Bit Rate [MBit/s]	2.5	5	10
gdMinislot^{Min}[MT]	2		
gNumberOfMinislots^{Max}[Minislot]	**3977**	**7977**	**7986**

Table B-19: Calculation of maximum values 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 7986 minislots.

To estimate the number of frames which could be transmitted in the dynamic segment the following equation may be useful:

[13] *aMinislotPerDynamicFrame*[Minislot] = 1 Minislot +

ceil(((*aFrameLengthDynamic*[gdBit] + *vDTSLow*^{Min}[gdBit]) * *gdBitMax*[µs/gdBit]) /

(*gdMacrotick*[µs/MT] * (1 - *cClockDeviationMax*) *

gdMinislot[MT/Minislot]) **)** + *gdDynamicSlotIdlePhase*[Minislot]

with

[14] *vDTSLow*^{Min} = 1 gdBit

B.4.15 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 *cClockDeviationMax*.

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. Detailed 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 22:

pRateCorrectionOut[µT] = **ceil(** *pMicroPerCycle*[µT] * 2 * *cClockDeviationMax* / (1 - *cClockDeviationMax*) **)**

Bit Rate [MBit/s]	2.5	5	10
pMicroPerCycle^{Min}[µT]	1040	640	656
pRateCorrectionOut^{Min}[µT]	**4**	**2**	**2**

Table B-20: Calculation of the minimum of *pRateCorrectionOut*.

***pdMicrotick*[µs]**	**0.100**	**0.050**	**0.025**	**0.0125**
pMicroPerCycle^{Max}[µT]	160000	320000	640000	

Table B-21: Calculation of the maximum of *pRateCorrectionOut*.

[117] That can be accomplished by e.g. increasing *gdNIT* or decreasing *gMacroPerCycle*.

***pdMicrotick*[µs]**	**0.100**	**0.050**	**0.025**	**0.0125**
pRateCorrectionOut^{Max}[µT]	**481**	**962**	**1923**	

Table B-21: Calculation of the maximum of *pRateCorrectionOut*.

As a result *pRateCorrectionOut* must be configurable over a range of 2 to 1923 µT.

B.4.16 Offset Correction

B.4.16.1 gOffsetCorrectionMax

The parameter *gOffsetCorrectionMax* represents the maximum amount of offset correction that would be required in a properly working system. Based on *gOffsetCorrectionMax* the node local parameter *pOffsetCorrectionOut* is configured. For *gOffsetCorrectionMax* two constraints must be fulfilled:

Constraint 23:

gOffsetCorrectionMax[µs] >= *gAssumedPrecision*[µs] + *gdMaxPropagationDelay*[µs] - *gdMinPropagationDelay*[µs]

Constraint 24:

gOffsetCorrectionMax[µs] <= *gdActionPointOffset*[MT] * *gdMacrotick*[µs/MT] * (1 + *cClockDeviationMax*) + *gdMinPropagationDelay*[µs] + *gdMaxPropagationDelay*[µs]

***gdMaxMicrotick*[µs]**	**0.100**	**0.050**	**0.025**	**0.0125**
gAssumedPrecision^{Min}[µs][a]	1.2	0.6	0.3	0.15
gdActionPointOffset^{Max}[MT]	63			
gdMacrotick^{Max}[µs]	6			3
gdMinPropagationDelay^{Min}[µs], *gdMaxPropagationDelay*^{Min}[µs]	0			
gdMinPropagationDelay^{Max}[µs], *gdMaxPropagationDelay*^{Max}[µs]	2.5			
gOffsetCorrectionMax^{Min}[µs]	**1.2**	**0.6**	**0.3**	**0.15**
gOffsetCorrectionMax^{Max}[µs]	**383.567**			**194.2835**

[a] *gAssumedPrecision*^{Min} is equal to *aBestCasePrecision*.

Table B-22: Calculation of the minimum and maximum of *gOffsetCorrectionMax*.

B.4.16.2 pOffsetCorrectionOut

Constraint 25:

pOffsetCorrectionOut[µT] = **ceil(** *gOffsetCorrectionMax*[µs] /

(*pdMicrotick*[µs/µT] * (1 - *cClockDeviationMax*)) **)**

***pdMicrotick*[µs]**	**0.100**	**0.050**	**0.025**	**0.0125**
$gOffsetCorrectionMax^{Min}$[µs]	1.2	0.6	0.3	0.15
$gOffsetCorrectionMax^{Max}$[µs]	383.567			194.2835
$pOffsetCorrectionOut^{Min}$[µT]	**13**			
$pOffsetCorrectionOut^{Max}$[µT]	**3842**	**7683**	**15366**	**15567**

Table B-23: Calculation of the minimum and maximum of *pOffsetCorrectionOut*.

As a result, the parameter *pOffsetCorrectionOut* shall be configurable over a range of 13 to 15567 µT.

B.4.17 gOffsetCorrectionStart

Consider the following assumptions:

- The offset correction phase starts at *gOffsetCorrectionStart*.
- The offset correction phase ends at the end of the cycle.

Thus it holds that

Constraint 26:

gOffsetCorrectionStart[MT] = *gMacroPerCycle*[MT] - *adOffsetCorrection*[MT]

with *adOffsetCorrection* the length of the offset correction phase, which is constrained in formula [17] in the next section.

B.4.18 gdNIT

Consider the following assumptions:

1. The NIT consists of offset calculation phase and offset correction phase.
2. The duration of offset calculation is implementation dependent and the phase must be completed before the start of the offset correction phase. The upper limit for the duration 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 *pOffsetCorrectionOut* resp. *gOffsetCorrectionMax*. 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 *cdMicroPerMacroMin* microticks.
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 phase is implementation dependent and must be completed before the end of the NIT. The upper limit is defined in section 8.6.3: The rate correction calculation must be com-

pleted 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 27:

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

[15] *adRemOffsetCalculation*[MT] <= **max(** 1, **ceil(** (*cdMaxOffsetCalculation*[µT] * *gdMaxMicrotick*[µs/µT] * (1 + *cClockDeviationMax*) - (*adActionPointDifference*[MT] + *gdMinislot*[MT] * *gNumberOfMinislots* + *gdSymbolWindow*[MT]) * *gdMacrotick*[µs] * (1 - *cClockDeviationMax*)) / (*gdMacrotick*[µs] * (1 - *cClockDeviationMax*)) **))**

[16] *adRemOffsetCalculation*[MT] >= 1

With item 5 *adOffsetCorrection*, the length of the offset correction phase, can be constrained by

[17] *adOffsetCorrection*[MT] >= **ceil(** *gOffsetCorrectionMax*[µs] / (*gdMacrotick*[µs/MT] * (1 - *cClockDeviationMax*) - *gdMaxMicrotick*[µs/µT] * *cMicroPerMacroMin*[µT/MT] * (1 + *cClockDeviationMax*)) **)**

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 such early that it finishes before the cycle start even if the maximum negative offset correction *gOffsetCorrectionMax* is applied, effectively adding *gOffsetCorrectionMax* to required calculation time. It is therefore constrained by

[18] *adRemRateCalculation*[MT] <= **max(** 1, **ceil(** (*cdMaxRateCalculation*[µT] * *gdMaxMicrotick*[µs/µT] * (1 + *cClockDeviationMax*) - (*adActionPointDifference*[MT] + *gdMinislot*[MT] * *gNumberOfMinislots* + *gdSymbolWindow*[MT]) * *gdMacrotick*[µs] * (1 - *cClockDeviationMax*) + *gOffsetCorrectionMax*[µs]) / (*gdMacrotick*[µs] * (1 - *cClockDeviationMax*)) **))**

Bit Rate [MBit/s]	2.5	5	10
gdNumberOfMinislots^{Min}	0		
gdSymbolWindow^{Min}[MT]	0		
gOffsetCorrectionMax^{Max}[µs]	383.567		
gdMaxMicrotick[µs]	0.05	0.025	0.025
gdMacrotick^{Min}[µT]	2	1	
adRemOffsetCalculation[MT]	34		
adRemRateCalculation[MT]	230	422	422

Table B-24: Calculation of the maximum of *gdNIT*.

Bit Rate [MBit/s]	2.5	5	10
adOffsetCorrection[MT]	386	771	771
$gdNIT^{Min}$[MT]	**2**	**2**	**2**
$gdNIT^{Max}$[MT]	**420**	**805**	**805**

Table B-24: Calculation of the maximum of *gdNIT*.

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.

As a result, the parameter *gdNIT* must be configurable over a range of 2 MT to 805 MT.

B.4.19 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 28:

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. The smaller the number of externally corrected microticks per double cycle is relative to the cycle length, the easier it is to achieve the required host synchronization. If the synchronization is done by using payload data bits to distribute the correction command {increase, no change, decrease} to all nodes, an absolute external correction value of less than 10 µT is preferred.

For this reason, the range of *pExternRateCorrection* shall be configurable over a range of 0 to 7 µT.

B.4.20 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*.

Constraint 29:

pExternOffsetCorrection[µT] <= *pOffsetCorrectionOut*[µT]

The range of *pExternOffsetCorrection* shall be configurable over a range of 0 to 7 µT.[118]

B.4.21 pdMaxDrift

Consider the following assumption:

- The maximum drift of cycle length between transmitting and receiving nodes is limited by twice the maximum deviation of the clock frequency *cClockDeviationMax*.

Constraint 30:

pdMaxDrift[µT] = **ceil(** *pMicroPerCycle*[µT] * 2 * *cClockDeviationMax* / (1 - *cClockDeviationMax*) **)**

[118] A small value for *pExternOffsetCorrection* (much smaller than *pOffsetCorrectionOut*) has the advantage that a node which does not apply the external offset correction by fault stays synchronized with the other nodes in the cluster.

***pdMicrotick*[µs]**	**0.100**	**0.050**	**0.025**	**0.0125**
pMicroPerCycle^{Max}[µT]	160000	320000	640000	
pdMaxDrift^{Max}[µT]	**481**	**962**	**1923**	

Table B-25: Calculation of the maximum of *pdMaxDrift*.

Bit Rate [MBit/s]	**2.5**	**5**	**10**
pMicroPerCycle^{Min}[µT]	1040	640	656
pdMaxDrift^{Min}[µT]	**4**	**2**	**2**

Table B-26: Calculation of the minimum of *pdMaxDrift*.

As a result, the parameter *pdMaxDrift* shall be configurable over a range of 2 to 1923 µT.

B.4.22 pdListenTimeout

To configure the parameter *pdListenTimeout* the following constraint must be taken into account:

Constraint 31:

pdListenTimeout[µT] = 2 * (*pMicroPerCycle*[µT] + *pdMaxDrift*[µT])

***pdMicrotick*[µs]**	**0.100**	**0.050**	**0.025**	**0.0125**
pMicroPerCycle^{Max}[µT]	160000	320000	640000	
pdMaxDrift^{Max}[µT]	481	962	1923	
pdListenTimeout^{Max}[µT]	**320962**	**641924**	**1283846**	

Table B-27: Calculation of the maximum of *pdListenTimeout*.

Bit Rate [MBit/s]	**2.5**	**5**	**10**
pMicroPerCycle^{Min}[µT]	1040	640	656
pdMaxDrift^{Min}[µT]	4	2	2
pdListenTimeout^{Min}[µT]	**2088**	**1284**	**1316**

Table B-28: Calculation of the minimum of *pdListenTimeout*.

As a result, the parameter *pdListenTimeout* shall be configurable over a range of 1284 to 1283846 µT.

B.4.23 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 3-10).

Constraint 32:

pDecodingCorrection[µT] = **round(** ((*gdTSSTransmitter*[gdBit] + *cdFSS*[gdBit] + 0.5 * *cdBSS*[gdBit]) * *cSamplesPerBit*[samples/gdBit] + *cStrobeOffset*[samples] + *cVotingDelay*[samples]) / *pSamplesPerMicrotick*[samples/µT] **)**

Bit Rate [MBit/s]	2.5	5	10
gdTSSTransmitter^{Min}[gdBit]	3	4	6
gdTSSTransmitter^{Max}[gdBit]	5	8	15
pSamplesPerMicrotick^{Min}	1		
pSamplesPerMicrotick^{Max}	2	4	
pDecodingCorrection^{Min}[µT]	**24**	**14**	**18**
pDecodingCorrection^{Max}[µT]	**63**	**87**	**143**

Table B-29: Calculation of minimum and maximum values of *pDecodingCorrection*.

As a result, the parameter *pDecodingCorrection* shall be configurable between 14 and 143 µT.

B.4.24 pMacroInitialOffset

Consider the following assumption:

- *pMacroInitialOffset[Ch]* has to be in the range of *gdActionPointOffset* < *pMacroInitialOffset[Ch]* < *gdStaticSlot*.

Constraint 33:

pMacroInitialOffset[Ch][MT] = *gdActionPointOffset*[MT] + **ceil(** (*pDecodingCorrection*[µT] + *pDelayCompensation[Ch]*[µT]) / *pMicroPerMacroNom*[µT/MT] **)**

Bit Rate [MBit/s]	2.5	5	10
gdActionPointOffset^{Min}[MT]	1		
gdActionPointOffset^{Max}[MT]	63		
pDecodingCorrection^{Min}[µT]	24	14	18
pDecodingCorrection^{Max}[µT]	63	87	143
pDelayCompensation[Ch]^{Min}[µT]	0		
pDelayCompensation[Ch]^{Max}[µT]	50	100	200
pMicroPerMacroNom^{Min}[µT]	40		80[a]
pMicroPerMacroNom^{Max}[µT]	60[b]	60[c]	120[d]
pMacroInitialOffset[Ch]^{Min}[MT]	**2**	**2**	**2**
pMacroInitialOffset[Ch]^{Max}[MT]	**66**	**68**	**68**

Table B-30: Calculation of minimum and maximum values of *pMacroInitialOffset[Ch]*.

[a] For the calculation *pMicroPerMacroNom*^{Min} = 80 is used because *pSamplesPerMicrotick* = 1 and *pMicroPerMacroNom*^{Min} = 40 are mutually exclusive for 10 MBit/s. Alternatively, *pMicroPerMacroNom*^{Min} = 40 could be used but then *pDecodingCorrection*^{Max} = 72 and *pDelayCompensation[Ch]*^{Max} = 100 must be used because of the longer necessary microtick. Both calculations lead to the same result documented in the table.

[b] For the calculation *pMicroPerMacroNom*^{Max} = 60 is used because *pSamplesPerMicrotick* = 2 and *pMicroPerMacroNom*^{Max} = 120 are mutually exclusive for 2.5 MBit/s.

[c] For the calculation *pMicroPerMacroNom*^{Max} = 60 is used because *pSamplesPerMicrotick* = 4 and *pMicroPerMacroNom*^{Max} = 240 are mutually exclusive for 5 MBit/s.

[d] For the calculation *pMicroPerMacroNom*^{Max} = 120 is used because *pSamplesPerMicrotick* = 4 and *pMicroPerMacroNom*^{Max} = 240 are mutually exclusive for 10 MBit/s.

As a result, the parameter *pMacroInitialOffset[Ch]* shall be configurable between 2 and 68 MT.

B.4.25 pMicroInitialOffset

Consider the following assumptions:

- *pMacroInitialOffset[Ch]*[MT] - *pMicroInitialOffset[Ch]*[µT] = secondary time reference point.
- 0 <= *pMicroInitialOffset[Ch]*[µT] <= *pMicroPerMacroNom*[µT]

Constraint 34:

pMicroInitialOffset[Ch][µT] = (*pMicroPerMacroNom*[µT] - ((*pDecodingCorrection*[µT] + *pDelayCompensation[Ch]*[µT]) **modulo** *pMicroPerMacroNom*[µT])) **modulo**[119] *pMicroPerMacroNom*[µT]

Because of the mechanisms defined in the startup the following constraint must be fulfilled:

Constraint 35:

pMicroInitialOffset[Ch][µT] < **floor(** (((5 + 2 * *gPayloadLengthStatic* + 3) * 10 - 2) * *gdBitMin*[µs]) / (*pdMicrotick*[µs/µT] * (1 + *cClockDeviationMax*)) **)**

Bit Rate [MBit/s]	2.5	5	10
pMicroPerMacroNom[µT]	112	186	240
pDecodingCorrection^{Max}[µT]	63	87	143
pDelayCompensation[Ch]^{Max}[µT]	50	100	200
pMicroInitialOffset[Ch]^{Min}[µT]	**0**	**0**	**0**
pMicroInitialOffset[Ch]^{Max}[µT]	**111**	**185**	**239**

Table B-31: Calculation of minimum and maximum values of *pMicroInitialOffset[Ch]*.

The parameter *pMicroInitialOffset[Ch]* shall be configurable over a range of 0 to 239 µT.

B.4.26 pLatestTx

Consider the following assumptions:

[119] The second modulo operation forces *pMicroInitialOffset[Ch]* to become zero if *pDecodingCorrection* + *pDelayCompensation[Ch]* is exactly a multiple of one macrotick.

- For a given node, the payload length of the longest dynamic frame for transmission is given by *aPayloadLengthDynamic* = *pPayloadLengthDynMax*[120].
- Each node must guarantee that even for the longest frame sent in the dynamic segment transmission is completed before the end of the dynamic segment.
- After each frame the dynamic slot idle phase must be taken into account.
- The influence of clock deviation (*cClockDeviationMax*) on the length of a macrotick must be taken into account.

Substituting the length of the maximum dynamic frame for *aPayloadLength* in equation [9] results in:

[19] *aFrameLengthDynamic* = *aFrameLength* with *aPayloadLength* = *pPayloadLengthDynMax*

With the definition of *aFrameLengthDynamic* in [19] and $vDTSLow^{Min}$ in [14], the constraint on *pLatestTx* is

Constraint 36:

pLatestTx[Minislot] <= **floor(** *gNumberOfMinislots*[Minislot] - (((*aFrameLengthDynamic*[gdBit] + $vDTSLow^{Min}$) * *gdBitMax*[µs/gdBit]) / (*gdMacrotick*[µs/MT] * (1 - *cClockDeviationMax*) * *gdMinislot*[MT/Minislot])) - *gdDynamicSlotIdlePhase*[Minislot] **)**

Bit Rate [MBit/s]	2.5	5	10
$gNumberOfMinislots^{Max}$[Minislot]	3977	7977	7986
$aFrameLengthDynamic^{Min}$[gdBit]	86	87	89
$vDTSLow^{Min}$[gdBit]	1		
gdBitMax[µs]	0.4006	0.2003	0.10015
$gdMacrotick^{Min}$[µs]	2	1	
$gdMinislot^{Min}$[MT]	2		
$gdDynamicSlotIdlePhase^{Min}$[Minislot]	1[a]		
$pLatestTx^{Max}$[Minislot]	**3967**	**7967**	**7980**

Table B-32: Calculation of the maximum values of *pLatestTx*.

[a] The minimum of *gdDynamicSlotIdlePhase* of zero as calculated in B.4.13 is not possible here because the minimum was calculated with the largest possible *gdMacrotick*. The calculation here is based on the smallest possible *gdMacrotick* which leads to a larger $gdDynamicSlotIdlePhase^{Min}$ value.

The minimum value would occur if given if the dynamic segment is only as long as the frame that must be transmitted plus *gdDynamicSlotIdlePhase*. In this case, the frame may start no later than the first minislot. It is desirable, however, to also use this parameter to allow a system designer to prevent a node from transmitting at all in the dynamic segment. Setting this parameter to zero would have this effect.

$pLatestTx^{Min}$ = 0 Minislot.

As a result, the parameter *pLatestTx* must be configurable over a range of 0 to 7980 minislots.

B.4.27 gdTSSTransmitter

Consider the following assumptions:

[120] 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.

- The transmission of a frame may be shortened at the beginning by an interval of up to *dStarTruncation*[µs][121] as the frame passes through an active star. The amount of truncation depends on the nodes 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.
- If a frame must pass through more than one star, each additional star causes additional truncation. For this reason the maximum number of active stars $nStarPath_{M,N}$ between any two nodes M and N must be taken into account.
- In addition to the effects described above, there is also some truncation that occurs in the BD of the receiving node (*dBDRxia*[µs][122]). This truncation is present even if the frame does not pass through any active stars.
- 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 37:

gdTSSTransmitter[gdBit] >= **ceil(** (*gdBitMax*[µs] + *dBDRxia*[µs] +

max({x | x = $nStarPath_{M,N}$ } **)** * *dStarTruncation*[µs]) / *gdBitMin*[µs/gdBit] **)**

Bit Rate [MBit/s]	2.5	5	10
gdBitMax[µs]	0.4006	0.2003	0.10015
gdBitMin[µs]	0.3994	0.1997	0.09985
$dStarTruncation^{Max}$[µs]	0.45		
$dBDRxia^{Max}$[µs]	0.45		
max({x | x = $nStarPath_{M,N}$} **)**^{Min}	0		
max({x | x = $nStarPath_{M,N}$} **)**^{Max}	2		
$gdTSSTransmitter^{Min}$[gdBit]	**3**	**4**	**6**
$gdTSSTransmitter^{Max}$[gdBit]	**5**	**8**	**15**

Table B-33: Calculation of minimum and maximum values of *gdTSSTransmitter*.

As a result, the parameter *gdTSSTransmitter* shall be configurable between 3 and 15 gdBit.

B.4.28 gdCASRxLowMax

The upper limit of the acceptance window for a collision avoidance symbol (CAS) must meet the following constraint:

Constraint 38:

gdCASRxLowMax[gdBit] = **ceil(** 2 * (*gdTSSTransmitter*[gdBit] + *cdCAS*[gdBit]) * (1 + *cClockDeviationMax*)

/ (1 - *cClockDeviationMax*) + 2 * *dBDRxai*[µs] / *gdBitMin*[µs/gdBit] **)**

[121] See the FlexRay Electrical Physical Layer Specification [EPL05].
[122] *dBDRxia* is the upper bound since the value *dBDTx01* (see [EPL05]) can be subtracted.

Bit Rate [MBit/s]	2.5	5	10
gdTSSTransmitter^{Min}[gdBit]	3	4	6
gdTSSTransmitter^{Max}[gdBit]	5	8	15
dBDRxai^{Min}[µs]	0		
dBDRxai^{Max}[µs]	0.4		
gdBitMin[µs]	0.3994	0.1997	0.09985
gdCASRxLowMax^{Min}[gdBit]	**67**	**69**	**73**
gdCASRxLowMax^{Max}[gdBit]	**73**	**81**	**99**

Table B-34: Calculation of values of *gdCASRxLowMax*.

As a result, the parameter *gdCASRxLowMax* shall be configurable between 67 and 99 gdBit.

B.4.29 gdWakeupSymbolTxIdle

The following constraint must be met:

Constraint 39:

gdWakeupSymbolTxIdle[gdBit] = **ceil(** *cdWakeupSymbolTxIdle*[µs] / *gdBit*[µs/gdBit] **)**

Bit Rate [MBit/s]	2.5	5	10
gdBit[µs]	0.4	0.2	0.1
gdWakeupSymbolTxIdle[gdBit]	**45**	**90**	**180**

Table B-35: Calculation of values of *gdWakeupSymbolTxIdle*.

As a result, the parameter *gdWakeupSymbolTxIdle* shall be configurable between 45 and 180 gdBit.

B.4.30 gdWakeupSymbolTxLow

The following constraint must be met:

Constraint 40:

gdWakeupSymbolTxLow[gdBit] = **ceil(** *cdWakeupSymbolTxLow*[µs] / *gdBit*[µs/gdBit] **)**

Bit Rate [MBit/s]	2.5	5	10
gdBit[µs]	0.4	0.2	0.1
gdWakeupSymbolTxLow[gdBit]	**15**	**30**	**60**

Table B-36: Calculation of values of *gdWakeupSymbolTxLow*.

As a result, the parameter *gdWakeupSymbolTxLow* shall be configurable between 15 and 60 gdBit.

B.4.31 gdWakeupSymbolRxIdle

Consider the following assumptions:

- Because of the clock deviation between nodes, the wakeup symbol seen by the receiver may be shorter than the transmitted wakeup symbol.

Constraint 41:

gdWakeupSymbolRxIdle[gdBit] = **floor(** (*gdWakeupSymbolTxIdle*[gdBit] * (1 - *cClockDeviationMax*) - *gdWakeupSymbolTxLow*[gdBit] * (1 + *cClockDeviationMax*)) / (2 * (1+ *cClockDeviationMax*)) **)**

Bit Rate [MBit/s]	2.5	5	10
gdWakeupSymbolTxIdle[gdBit]	45	90	180
gdWakeupSymbolTxLow[gdBit]	15	30	60
gdWakeupSymbolRxIdle[gdBit]	**14**	**29**	**59**

Table B-37: Calculation of values of *gdWakeupSymbolRxIdle*.

As a result, the parameter *gdWakeupSymbolRxIdle* shall be configurable between 14 and 59 gdBit.

B.4.32 gdWakeupSymbolRxLow

Consider the following assumptions:

- Because of the clock deviation between nodes, the wakeup symbol seen by the receiver may be shorter than the transmitted wakeup symbol.
- The truncation of the low level of a wakeup symbol is taken into account by the parameters *dStarTruncation* and *dBDRxia*[123].

Constraint 42:

gdWakeupSymbolRxLow[gdBit] = **floor(** (*gdWakeupSymbolTxLow*[gdBit] * *gdBitMin*[µs/gdBit] - (**max(** {x | x = $nStarPath_{M,N}$ } **)** * *dStarTruncation*[µs] + *dBDRxia*[µs])) / *gdBitMax*[µs/gdBit] **)**

Bit Rate [MBit/s]	2.5	5	10
gdWakeupSymbolTxLow[gdBit]	15	30	60
gdBitMin[µs]	0.3994	0.1997	0.09985
gdBitMax[µs]	0.4006	0.2003	0.10015
$dStarTruncation^{Max}$[µs]	0.45		
$dBDRxia^{Min}$[µs]	0		

Table B-38: Calculation of minimum and maximum values of *gdWakeupSymbolRxLow*.

[123] *dBDRxia* is the upper bound since the value *dBDTx01* (see [EPL05]) can be subtracted.

Bit Rate [MBit/s]	2.5	5	10
dBDRxia^{Max}[µs]	0.45		
max({x | x = *nStarPath*$_{M,N}$} **)**^{Min}	0		
max({x | x = *nStarPath*$_{M,N}$} **)**^{Max}	2		
gdWakeupSymbolRxLow^{Min}[gdBit]	**11**	**23**	**46**
gdWakeupSymbolRxLow^{Max}[gdBit]	**14**	**29**	**59**

Table B-38: Calculation of minimum and maximum values of *gdWakeupSymbolRxLow*.

As a result, the parameter *gdWakeupSymbolRxLow* shall be configurable between 11 and 59 gdBit.

B.4.33 gdWakeupSymbolRxWindow

Consider the following assumptions:

- Because of the clock deviation between nodes, the wakeup symbol seen by the receiver may be longer than the transmitted wakeup symbol.

Constraint 43:

gdWakeupSymbolRxWindow[gdBit] = **ceil(** (2 * *gdWakeupSymbolTxLow*[gdBit] + *gdWakeupSymbolTxIdle*[gdBit]) * (1+ *cClockDeviationMax*) / (1 - *cClockDeviationMax*) **)**

Bit Rate [MBit/s]	2.5	5	10
gdWakeupSymbolTxLow[gdBit]	15	30	60
gdWakeupSymbolTxIdle[gdBit]	45	90	180
gdWakeupSymbolRxWindow[gdBit]	**76**	**151**	**301**

Table B-39: Calculation of values of *gdWakeupSymbolRxWindow*.

As a result, the parameter *gdWakeupSymbolRxWindow* shall be configurable between 76 and 301 gdBit.