INTERNATIONAL STANDARD

**ISO 17458-2**

First edition
2013-02-01

# Road vehicles— FlexRay communications system —

## Part 2:
## Data link layer specification

*Véhicules routiers — Système de communications FlexRay —*

*Partie 2: Spécification de la couche de liaison de données*



Reference number
ISO 17458-2:2013(E)



ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

# Contents

Page

# Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO 17458-2 was prepared by Technical Committee ISO/TC 22, *Road vehicles*, Subcommittee SC 3, *Electrical and electronic equipment*.

ISO 17458 consists of the following parts, under the general title *Road vehicles — FlexRay communications system*:

— *Part 1: General information and use case definition*

— *Part 2: Data link layer specification*

— *Part 3: Data link layer conformance test specification*

— *Part 4: Electrical physical layer specification*

— *Part 5: Electrical physical layer conformance test specification*

# Introduction

The FlexRay communications system is an automotive focused high speed network and was developed with several main objectives which were defined beyond the capabilities of established standardized bus systems like CAN and some other proprietary bus systems. Some of the basic characteristics of the FlexRay protocol are synchronous and asynchronous frame transfer, guaranteed frame latency and jitter during synchronous transfer, prioritization of frames during asynchronous transfer, single or multi-master clock synchronisation, time synchronisation across multiple networks, error detection and signalling, and scalable fault tolerance.

The FlexRay communications system is defined for advanced automotive control applications. It serves as a communication infrastructure for future generation high-speed control applications in vehicles by providing:

— A message exchange service that provides deterministic cycle based message transport;

— Synchronisation service that provides a common time base to all nodes;

— Start-up service that provides an autonomous start-up procedure;

— Error management service that provides error handling and error signalling;

— Wakeup service that addresses the power management needs;

Since start of development the automotive industry world wide supported the specification development. The FlexRay communications system has been successfully implemented in production vehicles today.

The ISO 17458 series specifies the use cases, the communication protocol and physical layer requirements of an in-vehicle communication network called "FlexRay communications system".

This part of ISO 17458 has been established in order to define the protocol requirements for vehicle communication systems implemented on a FlexRay data link.

To achieve this, it is based on the Open Systems Interconnection (OSI) Basic Reference Model specified in ISO/IEC 7498-1 and ISO/IEC 10731, which structures communication systems into seven layers. When mapped on this model, the protocol and physical layer requirements specified by ISO 17458 are broken into:

— Diagnostic services (layer 7), specified in ISO 14229-1 [7], ISO 14229-4 [9];

— Presentation layer (layer 6), vehicle manufacturer specific;

— Session layer services (layer 5), specified in ISO°14229-2 [8];

— Transport layer services (layer 4), specified in ISO 10681-2 [5];

— Network layer services (layer 3), specified in ISO 10681-2 [5];

— Data link layer (layer 2), specified in ISO 17458-2, ISO 17458-3;

— Physical layer (layer 1), specified in ISO 17458-4, ISO 17458-5;

in accordance with Table 1.

**Table 1 — FlexRay communications system specifications applicable to the OSI layers**

| Applicability | OSI 7 layers | FlexRay communications system | Vehicle manufacturer enhanced diagnostics |
|---|---|---|---|
| Seven layer according to ISO 7498-1 and ISO/IEC 10731 | Application (layer 7) | vehicle manufacturer specific | ISO 14229-1, ISO 14229-4 |
| | Presentation (layer 6) | vehicle manufacturer specific | vehicle manufacturer specific |
| | Session (layer 5) | vehicle manufacturer specific | ISO 14229-2 |
| | Transport (layer 4) | vehicle manufacturer specific | ISO 10681-2 |
| | Network (layer 3) | vehicle manufacturer specific | |
| | Data link (layer 2) | ISO 17458-2, ISO 17458-3 | |
| | Physical (layer 1) | ISO 17458-4, ISO 17458-5 | |

Table 1 shows ISO 17458 Parts 2 – 5 being the common standards for the OSI layers 1 and 2 for the FlexRay communications system and the vehicle manufacturer enhanced diagnostics.

The FlexRay communications system column shows vehicle manufacturer specific definitions for OSI layers 3 – 7.

The vehicle manufacturer enhanced diagnostics column shows application layer services covered by ISO 14229-4 which have been defined in compliance with diagnostic services established in ISO 14229-1, but are not limited to use only with them. ISO 14229-4 is also compatible with most diagnostic services defined in national standards or vehicle manufacturer's specifications. The presentation layer is defined vehicle manufacturer specific. The session layer services are covered by ISO 14229-2. The transport protocol and network layer services are specified in ISO 10681.

# Road vehicles — FlexRay communications system — Part 2: Data link layer specification

## 1 Scope

This part of ISO 17458 specifies the FlexRay communication protocol which is specified for a dependable automotive network. Some of the basic characteristics of the FlexRay protocol are synchronous and asynchronous frame transfer, guaranteed frame latency and jitter during synchronous transfer, prioritization of frames during asynchronous transfer, single or multi-master clock synchronisation [1)] time synchronisation across multiple networks, error detection and signalling, and scalable fault tolerance [2)].

## 2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 17458-1, *Road vehicles — FlexRay communications system — Part 1: General information and use case definition*

## 3 Terms, definitions, symbols and abbreviated terms

### 3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO 17458-1 and the following apply.

**3.1.1**
**application data**
data produced and / or used by application tasks

NOTE In the automotive context the term 'signal' is often used for application data exchanged among tasks.

**3.1.2**
**bus**
communication system topology in which nodes are directly connected to a single, common communication media (as opposed to connection through stars, gateways, etc.)

NOTE The term bus is also used to refer to the media itself.

1) Multi-master clock synchronisation refers to a synchronisation that is based on the clocks of several (three or more) synchronisation 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 (i.e., single or dual channel clusters, clusters with many or few sync nodes, etc.).

**3.1.3**
**channel idle**
condition on the physical transmission medium when no node is transmitting, as perceived by each individual node in the network

NOTE The detection of channel idle occurs some time after all nodes have actually stopped transmitting (due to idle detection times, channel effects, ringing, etc.).

**3.1.4**
**clique**
set of communication controllers having the same view of certain systems properties

EXAMPLE The global time value or the activity state of communication controllers.

**3.1.5**
**cluster**
communication system of multiple nodes connected via at least one communication channel directly (bus topology), by active stars (star topology) or by a combination of bus and star connections (hybrid topologies)

NOTE Clusters can be coupled by gateways.

**3.1.6**
**coldstart node**
node capable of initiating the communication startup procedure on the cluster by sending startup frames

NOTE TT-D coldstart nodes, TT-L coldstart nodes, and TT-E coldstart nodes are all considered to be coldstart nodes. By definition, all coldstart nodes are also sync nodes.

**3.1.7**
**communication slot**
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

NOTE FlexRay distinguishes between static communication slots and dynamic communication slots.

**3.1.8**
**cycle-dependent slot assignment**
method of assigning, for a given channel, an individual slot (identified by a specific slot number and a specific cycle counter number) or a set of slots (identified by a specific slot number and a set of communication cycle numbers) to a node

**3.1.9**
**cycle-independent slot assignment**
method of assigning, for a given channel, the set of all communication slots having a specific slot number to a node (i.e., on the given channel, slots with the specific slot number are assigned to the node in all communication cycles)

**3.1.10**
**cycle number**
positive integer used to identify a communication cycle

NOTE The cycle number of each communication cycle is one greater than the cycle number of the previous cycle, except in cases where the previous cycle had the maximum cycle number value, in which case the cycle number has the value of zero. The cycle number of the first cycle is, by definition, zero.

**3.1.11**
**cycle time**
time within the current communication cycle, expressed in units of macroticks

NOTE Cycle time is reset to zero at the beginning of each communication cycle.

**3.1.12**
**dynamic segment**
portion of the communication cycle where the media access is controlled via a mini-slotting scheme

NOTE 1 During this segment access to the media is dynamically granted on a priority basis to nodes with data to transmit.

NOTE 2 Also known as Flexible Time Division Multiple Access (FTDMA).

**3.1.13**
**dynamic slot / dynamic communication slot**
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

NOTE 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.

**3.1.14**
**frame**
structure used by the communication system to exchange information within the system

NOTE A frame consists of a header segment, a payload segment and a trailer segment. The payload segment is used to convey application data.

**3.1.15**
**frame identifier**
slot position in the static segment and priority in the dynamic segment

NOTE A lower identifier indicates a higher priority.

**3.1.16**
**global time**
combination of cycle counter and cycle time

**3.1.17**
**Hamming distance**
minimum distance (i.e., the number of bits which differ) between any two valid code words in a binary code

**3.1.18**
**implementation dependent**
behaviour that, subject to restrictions in the specification, may be chosen by an implementation designer. Implementation dependent behaviour shall be described in detail in the documentation of an implementation

**3.1.19**
**key slot**
static slot that is used by a node to transmit sync and startup frames or the slot that is used to transmit when the node is operating in key slot only mode.

**3.1.20**
**macrotick**
interval of time derived from the cluster-wide clock synchronisation algorithm

NOTE A macrotick consists of an integral number of microticks. The actual number of microticks in a given macrotick is adjusted by the clock synchronisation algorithm. The macrotick represents the smallest granularity unit of the global time.

**3.1.21**
**microtick**
interval of time derived directly from the CC's oscillator (possibly through the use of a prescaler)

NOTE The microtick is not affected by the clock synchronisation mechanisms, and is thus a node-local concept. Different nodes can have microticks of different duration.

**3.1.22**
**minislot**
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

**3.1.23**
**non-coldstart node**
node that is not capable of initiating the communication startup procedure (i.e., does not transmit startup frames)

**3.1.24**
**non-sync node**
node that is not configured to transmit sync frames.

**3.1.25**
**non-synchronized operation**
operation of a node when the node does not have a notion of FlexRay time, i.e., has no knowledge of slot identifier, slot boundaries, cycle counter, or segment boundaries

**3.1.26**
**network**
combination of the communication channels that connect the nodes of a cluster

**3.1.27**
**node**
logical entity connected to the network that is capable of sending and / or receiving frames

**3.1.28**
**null frame**
frame that contains no usable data in the payload segment

NOTE A null frame is indicated by a bit in the header segment, and all data bytes in the payload segment are set to zero.

**3.1.29**
**physical communication link**
inter-node connection through which signals are conveyed for the purpose of communication

NOTE 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.

**3.1.30**
**precision**
worst-case deviation between the corresponding macroticks of any two synchronized nodes in the cluster

**3.1.31**
**slot**
see communication slot

**3.1.32**
**slot ID (identifier)**
see slot number

**3.1.33**
**slot multiplexing**
technique of assigning, for a given channel, slots having the same slot identifier to different nodes in different communication cycles

**3.1.34**
**slot number**
number used to identify a specific slot within a communication cycle

**3.1.35**
**star**
device that allows information to be transferred from one physical communication link to one or more other physical communication links

NOTE A star duplicates information present on one of its links to the other links connected to the star. A star can be either passive or active. For the purposes of this specification, all usages of the term "star" are references to an active star as described in ISO 17458-4.

**3.1.36**
**startup frame**
FlexRay frame whose header segment contains an indicator that integrating nodes may use timerelated information from this frame for initialisation during the startup process

NOTE Startup frames are always also sync frames.

**3.1.37**
**static slot / static communication slot**
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

NOTE 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.

**3.1.38**
**static segment**
portion of the communication cycle where the media access is controlled via a static Time Division Multiple Access (TDMA) scheme

NOTE During this segment access to the media is determined solely by the progression of time.

**3.1.39**
**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 synchronisation algorithm

**3.1.40**
**sync node**
node configured to transmit sync frames

NOTE Coldstart nodes and TT-D non-coldstart sync nodes are considered to be sync nodes.

**3.1.41**
**synchronized operation**
operation of a node when the node has a notion of FlexRay time, i.e., has knowledge of slot identifier, slot boundaries, cycle counter, and segment boundaries

**3.1.42**
**time gateway**
pair of nodes attached to different clusters connected by a time gateway interface

**3.1.43**
**time gateway interface**
interface used by a time gateway source node to provide timing information for a time gateway sink node

**3.1.44**
**time gateway sink node**
node configured as TT-E coldstart node, which is connected via a time gateway interface to a time gateway source node

NOTE The time gateway sink node receives timing information from the time gateway source node.

**3.1.45**
**time gateway source node**
node connected via a time gateway interface to a time gateway sink node

NOTE The time gateway source node provides timing information for the time gateway sink node.

**3.1.46**
**time sink cluster**
cluster using the TT-E synchronisation method

NOTE The term emphasizes that the TT-E coldstart nodes of this cluster receive their timing from another cluster.

**3.1.47**
**time source cluster**
cluster that provides the timing information for a time sink cluster

**3.1.48**
**transmission slot assignment list**
structure identifying the set of all slots assigned to a node for transmission

**3.1.49**
**TT-D cluster**
cluster in which the clock synchronisation uses the TT-D synchronisation method

NOTE A TT-D cluster consists of two or more TT-D coldstart nodes, zero or more TT-D non-coldstart sync nodes and, zero or more non-sync nodes.

**3.1.50**
**TT-D coldstart node**
coldstart node operating in a TT-D cluster

NOTE This node has only a single key slot and sends a startup / sync frame in the configured key slot in each cycle on each configured channel.

**3.1.51**
**TT-D non-coldstart sync node**
node that is configured to transmit sync frames but is not capable of initiating the communication startup procedure (i.e., does not send startup frames)

**3.1.52**
**TT-D synchronisation method**
method of clock synchronisation in which the clock synchronisation is derived in a distributed manner from two or more sync nodes

NOTE Two or more coldstart nodes are required to start up a cluster using this synchronisation method.

**3.1.53**
**TT-E cluster**
cluster in which the clock synchronisation uses the TT-E synchronisation method



NOTE A TT-E cluster consists of one or more TT-E Coldstart nodes and zero or more non-sync nodes.

**3.1.54**
**TT-E coldstart node**
coldstart node operating in a TT-E cluster

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

**3.1.55**
**TT-E synchronisation method**
method of clock synchronisation in which the clock synchronisation is derived directly from the clock synchronisation of another FlexRay cluster

NOTE In this method a single coldstart node is capable of starting up the cluster.

**3.1.56**
**TT-L cluster**
cluster in which the clock synchronisation uses the TT-L synchronisation method

NOTE A TT-L cluster consists of one TT-L Coldstart node and one or more non-sync nodes.

**3.1.57**
**TT-L coldstart node**
coldstart node operating in a TT-L cluster

NOTE This node has two key slots and sends startup / sync frames in both configured key slots in each cycle on each configured channel.

**3.1.58**
**TT-L synchronisation method**
method of clock synchronisation in which the clock synchronisation is derived from the local clock of a single sync node, and in which a single coldstart node starts up the cluster

## 3.2 Symbols

∑ Summation symbol, a large upright capital Sigma

∈ Element, lower-case epsilon

∀ For all (given any), universal quantifier symbol, a turned "A"

## 3.3 Abbreviated terms

| | |
|---|---|
| µs | Microsecond |
| µT | Microtick |
| AP | Action Point |
| BD | Bus Driver |
| BIST | Built-In Self Test |
| BITSTRB | Bit Strobing Process |

| | |
|---|---|
| BSS | Byte Start Sequence |
| CAS | Collision Avoidance Symbol |
| CC | Communication Controller |
| CE | Communication Element |
| CHI | Controller Host Interface |
| CHIRP | Channel Idle Recognition Point |
| CODEC | Coding and Decoding Process |
| CRC | Cyclic Redundancy Code |
| CSP | Clock Synchronisation Process |
| CSS | Clock Synchronisation Startup Process |
| DTS | Dynamic Trailing Sequence |
| ECU | Electronic Control Unit, same as node |
| EMI | Electromagnetic Interference |
| ERRN | Error Not signal |
| FES | Frame End Sequence |
| FIFO | First In First Out |
| FSP | Frame and Symbol Processing |
| FSS | Frame Start Sequence |
| FTDMA | Flexible Time Division Multiple Access |
| FTM | Fault-Tolerant Midpoint |
| ID | Identifier |
| INH1 | Inhibit signal |
| MAC | Media Access Control Process |
| MT | Macrotick |
| MTG | Macrotick Generation Process |
| MTS | Media Access Test Symbol |
| NIT | Network Idle Time |
| NM | Network Management |
| POC | Protocol Operation Control |
| RxD | Receive data signal from bus driver |

| | |
|---|---|
| SDL | Specification and Description Language |
| SPI | Serial Peripheral Interface |
| STBN | Standby Not signal |
| SW | Symbol Window |
| sync | synchronisation |
| TDMA | Time Division Multiple Access |
| TRP | Time Reference Point |
| TSS | Transmission Start Sequence |
| TT-D | Time-Triggered Local Distributed |
| TT-E | Time-Triggered External |
| TT-L | Time-Triggered Local Master |
| TxD | Transmit Data signal from CC |
| TxEN | Transmit Data Enable Not signal from CC |
| WUDOP | Wakeup During Operation Pattern |
| WUP | Wakeup Pattern |
| WUPDEC | Wakeup Pattern Decoding Process |
| WUS | Wakeup Symbol |

## 4 Document overview

Figure 1 illustrates the document references.

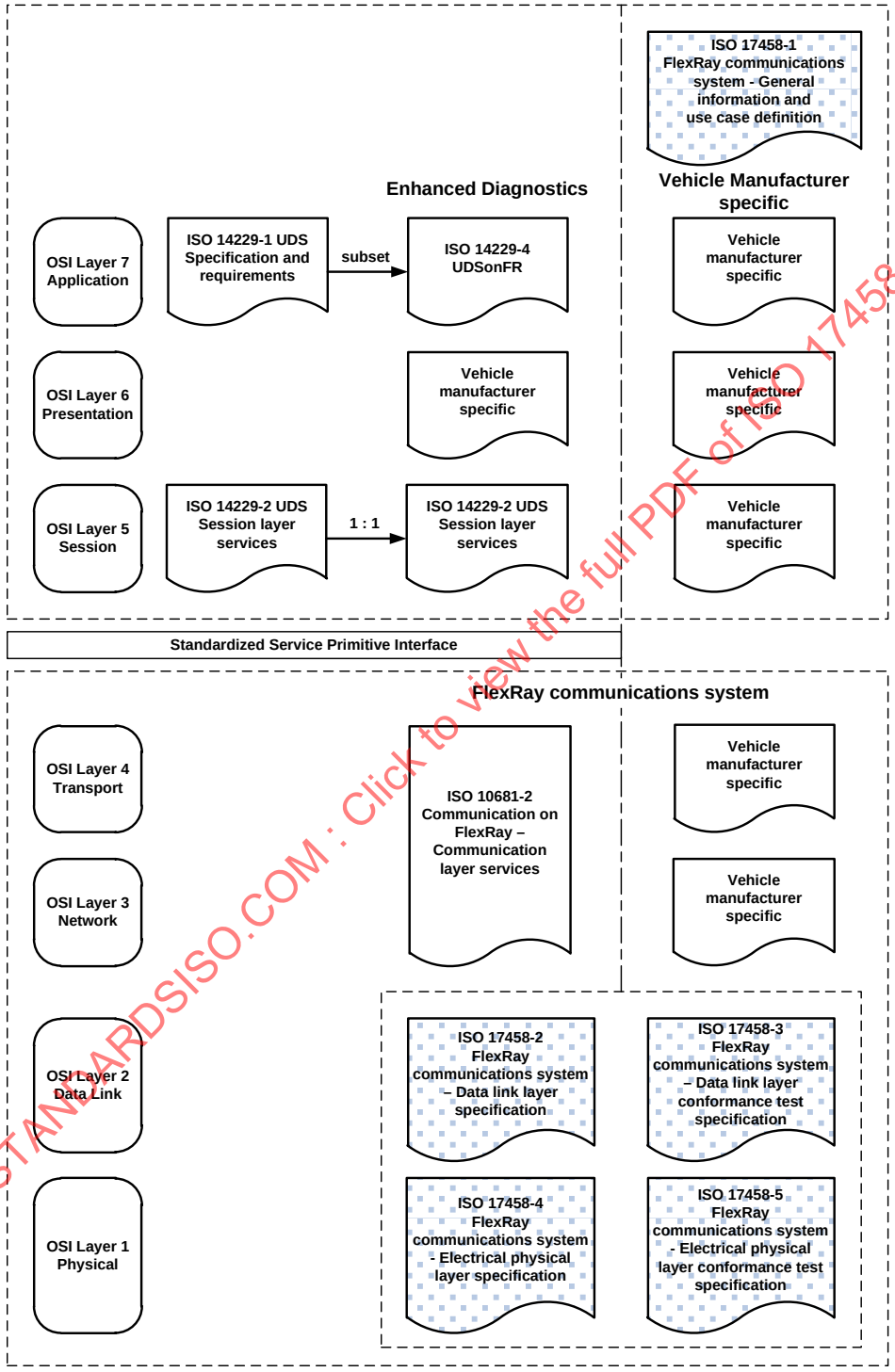





**Figure 1 — FlexRay document reference according to OSI model**

# 5 Conventions

## 5.1 General

ISO 17458 are based on the conventions specified in the OSI Service Conventions (ISO/IEC 10731) as they apply for physical and data link layer (protocol).

## 5.2 Notational conventions

### 5.2.1 Parameter prefix conventions

The following is a list of parameter prefix conventions:

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

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

— <prefix_2>::= d | s;

Table 2 defines the parameter <prefix_1>.

**Table 2 — Definition of parameter <prefix_1>**

| 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 shall 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 behaviour. 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 3 defines the parameter <prefix_2>.

**Table 3 — Definition of parameter <prefix_2>**

| 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.). |

### 5.2.2 Text coding

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

Parameters, constants and variables are highlighted in *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 5.2.1.

SDL states are also highlighted in *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 5.3.2.

SDL signals are also highlighted in *italics*. An example is the SDL signal *CHIRP on A*. 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.

### 5.2.3 Implementation dependent behaviour

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

— a behaviour (or a parameter or characterization of a behaviour, such as a default value) that, subject to restrictions contained in this specification, may be chosen by an implementation designer;

— implementation dependent behaviour may vary from implementation to implementation, but the specific behaviour shall be described in detail in the documentation of the implementation.

## 5.3 SDL conventions

### 5.3.1 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 [10]. 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 behavioural 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 behaviour 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 behaviour depicted in the SDL diagrams. Some technical details are intentionally omitted from these explanations. Unless specifically mentioned, the behaviour depicted in the SDL diagrams takes precedence over any textual description.

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

### 5.3.2 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.

### 5.3.3 SDL extensions

#### 5.3.3.1 General

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 [10]. This subclause documents significant deviations from "standard" SDL.

#### 5.3.3.2 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 clause 12 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 behaviour 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. 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 behaviour 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 [10] 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)]

#### 5.3.3.3 Microtick behaviour of the 'now' – expression

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

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

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

#### 5.3.3.5 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 behaviour 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

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

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.

#### 5.3.3.6 Signals to non-instantiated processes

In various portions of the SDL behavioural descriptions the SDL sends signals that are received by a process that in some conditions may not be instantiated. As an example, the SDL in Figure 30 generates the signal *CODEC control on B (NORMAL)* even if the node is only attached to channel A (implying that the CODEC_B process is not instantiated). The sending of these signals should not be interpreted as requiring that processes that receive the signal should be instantiated - in such cases these signals should simply be ignored. This convention holds even if the only process that consumes the signal in question is a process that is not already instantiated.

#### 5.3.3.7 Exported and imported signals

Certain features of the FlexRay protocol require that a certain direct communication between the two communication controllers of a time gateway is modeled within the SDL diagrams (for example, see 5.6.4). Signals marked with the EXP keyword are distributed within the local communication controller like any other signal, but are in addition are also forwarded to a second communication controller that is represented by a separate instance of the SDL diagrams. On the receiving end, these exported signals can be received by input symbols marked with the IMP keyword. Input symbols marked in this way are only sensitive to signals emitted with the EXP keyword of the other communication controller.

## 5.4 Bit rates

The FlexRay Communications System specifies three standard bit rates – 10 Mbit / s (corresponding to a nominal bit duration, *gdBit*, of 100 ns), 5 Mbit / s (corresponding to a *gdBit* of 200 ns), and 2,5 Mbit / s (corresponding to a *gdBit* of 400 ns). In order to be considered FlexRay conformant, a protocol implementation is required to support all three standard bit rates.

## 5.5 Roles of a node in a FlexRay cluster

There are three distinct roles a node can perform.

— The role of a sync node enables a node to actively participate in the clock synchronisation algorithm performed by the cluster. Sync nodes transmit sync frames that are evaluated by all nodes of the cluster to perform an alignment of clock rate, that effectively determines the cycle length, and clock offset, that effectively determines the position of the cycle start.

— The role of a coldstart node enables a node to initiate the communication. Coldstart nodes are allowed to start transmitting startup frames in the non-synchronized state with the intent of establishing a schedule. Nodes integrate onto that new schedule by evaluating the content and timing of the received startup frames. A coldstart node is always also a sync node and a startup frame is always also a sync frame.

— A node that is neither a coldstart node nor a sync node is referred to as non-sync node. It performs no special task.

## 5.6 Synchronisation methods

### 5.6.1 General

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

### 5.6.2 TT-D synchronisation method

A cluster in which the coldstart nodes use the TT-D synchronisation method is a TT-D cluster. The TT-D synchronisation method uses a distributed algorithm to reduce the effect of any single failure. No critical task depends on any single node. A distributed startup instigated and carried through by two to fifteen coldstart nodes mitigates many adverse effects a single faulty coldstart node can have (see clause 11). A distributed clock synchronisation algorithm actively driven by two to fifteen sync nodes is robust against a number of Byzantine faults depending on the number of currently active sync nodes (see clause 12).

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

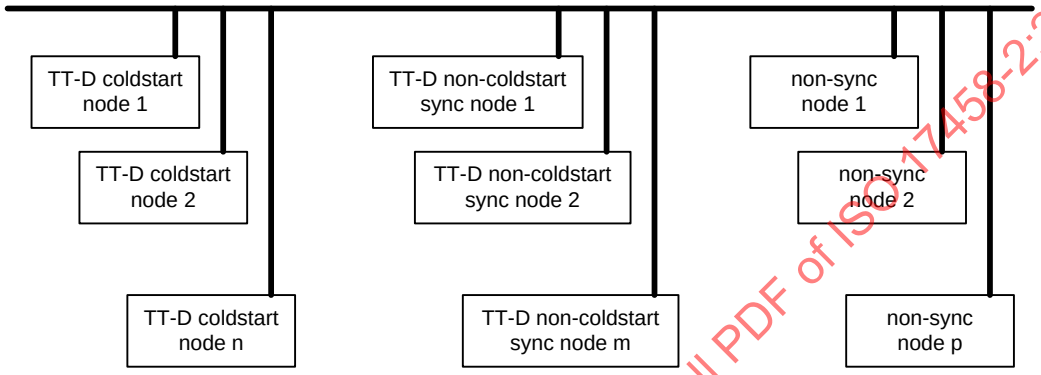



**Figure 2 — TT-D cluster**

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

### 5.6.3 TT-L synchronisation method

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

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

Figure 3 shows the configuration of nodes in a TT-L cluster. There exists exactly one TT-L coldstart node. The number of non-sync nodes p is not limited by the protocol.

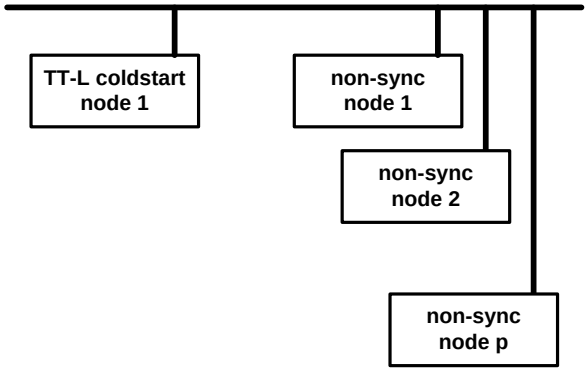



**Figure 3 — TT-L cluster**

### 5.6.4 TT-E synchronisation method

A cluster in which the coldstart nodes use the TT-E synchronisation method is a TT-E cluster. The primary intent of the TT-E synchronisation method is to synchronize the schedule of the TT-E cluster, also called a time sink cluster, to a second FlexRay cluster, which is referred to as time source cluster. To that end, each TT-E coldstart node, also called a time gateway sink node, shall be paired with a node of its time source cluster; this pair of nodes is called a time gateway. The node on the time sink cluster side is then called a time gateway sink node while the node on the time source cluster side is called time gateway source node.

Figure 4 depicts the basic setup. Depending on the synchronisation method employed by time source cluster, the time gateway source node may be a TT-D coldstart node, a TT-D non-coldstart sync node, a TT-L coldstart node, or a non-sync node[6)].

The two nodes of the time gateway are connected via a time gateway interface, which is used by the time gateway source node to provide information about the schedule of the time source cluster to the time gateway sink node.

Instead of the usual distributed startup, a TT-E coldstart node derives the cycle length and position of the cycle start from its time gateway source node and directly starts transmitting according to this schedule (slightly shifted with a fixed offset of *cdTSrcCycleOffset* microticks).

Similar to the TT-L synchronisation method, each TT-E coldstart node transmits two startup frames, so that a single TT-E coldstart node suffices to start and maintain a TT-E cluster. Contrary to the TT-L synchronisation method, multiple TT-E coldstart nodes may be present in a TT-E cluster. As all TT-E coldstart nodes derive their schedule from the same cluster, they are implicitly synchronized to one another.



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

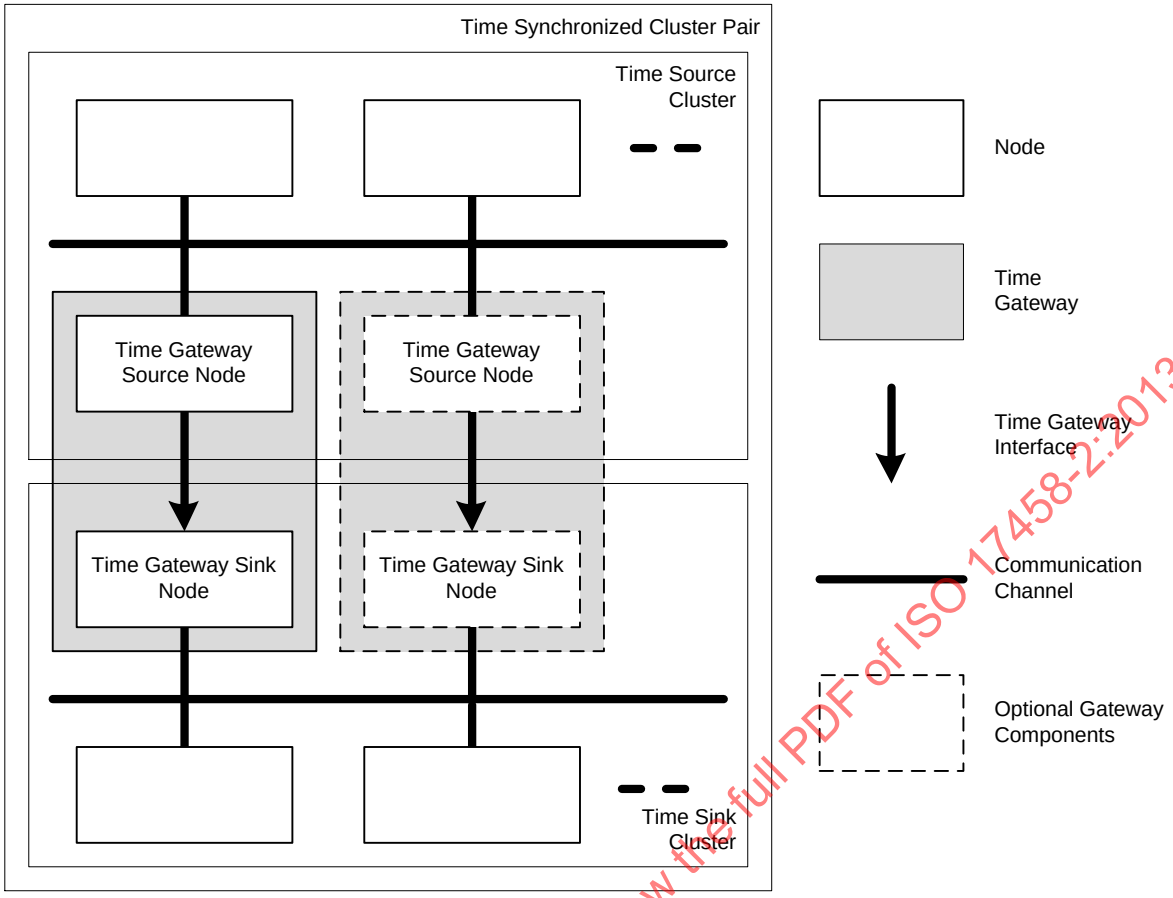



**Figure 4 — Time synchronized cluster pair**

The advantage of the TT-E synchronisation method is the close coupling of the schedule of a TT-E cluster to another FlexRay cluster. In this way, a single FlexRay cluster may be split into synchronized sub-clusters to avoid limits on attached nodes placed upon a single FlexRay cluster by ISO 17458-4 or to enable a separation of nodes into multiple clusters according to communication needs for a more efficient use of the available bandwidth.

Figure 5 shows the configuration of two connected clusters, the lower being a time sink cluster, the upper being a time source cluster, in this case a TT-D cluster. The TT-D cluster could also be replaced by a TT-L cluster or TT-E cluster[7]. The number of TT-E coldstart nodes 'i' shall be at least one and less than or equal to 7. The number of non-sync nodes 'k' is not limited by the protocol.

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



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

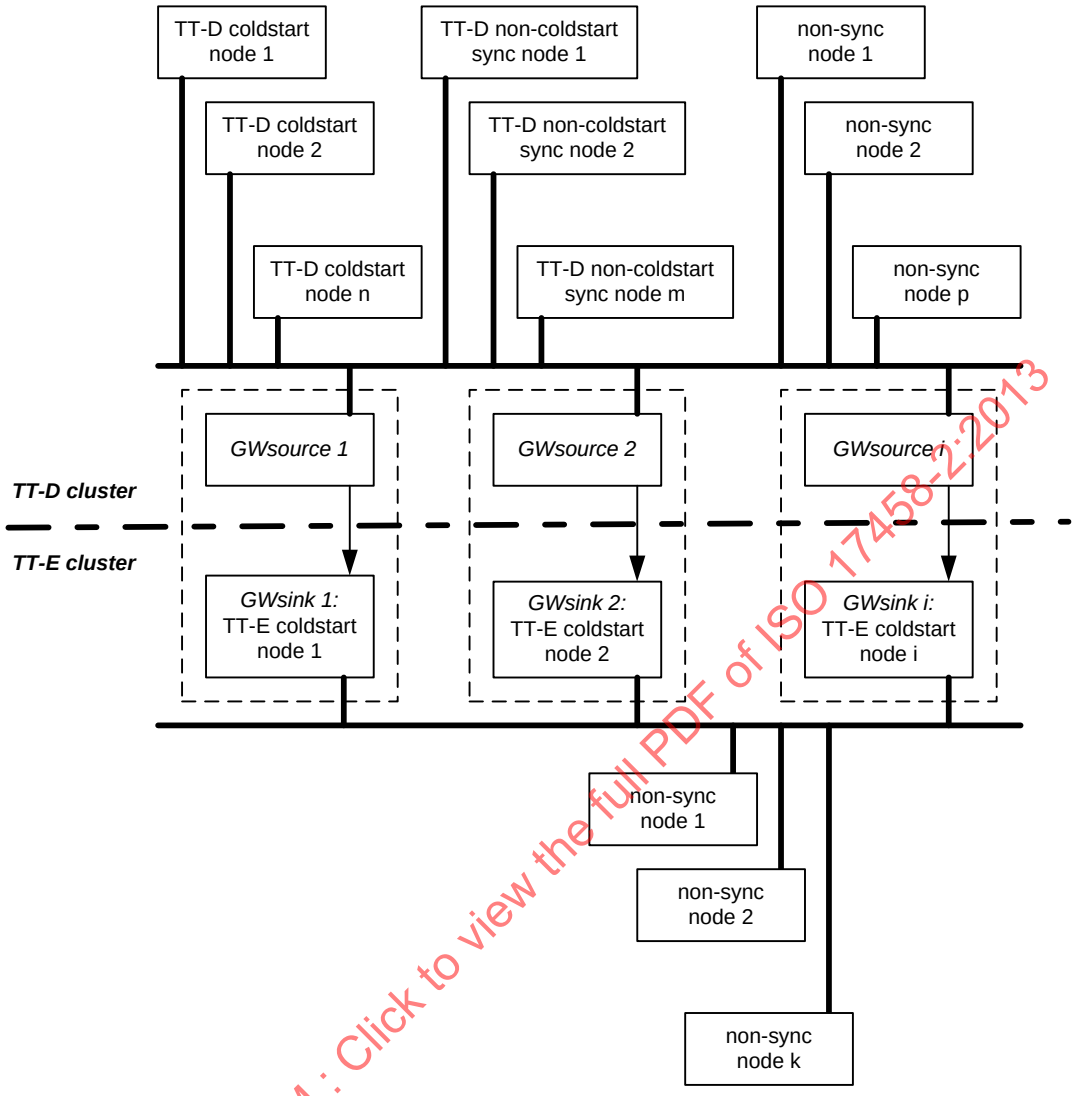



**Figure 5 — TT-E cluster**

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

## 5.7 Network topology considerations

### 5.7.1 General

The following subclauses 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 ISO 17458-4.

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

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

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

### 5.7.2 Passive bus topology

Figure 6 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).

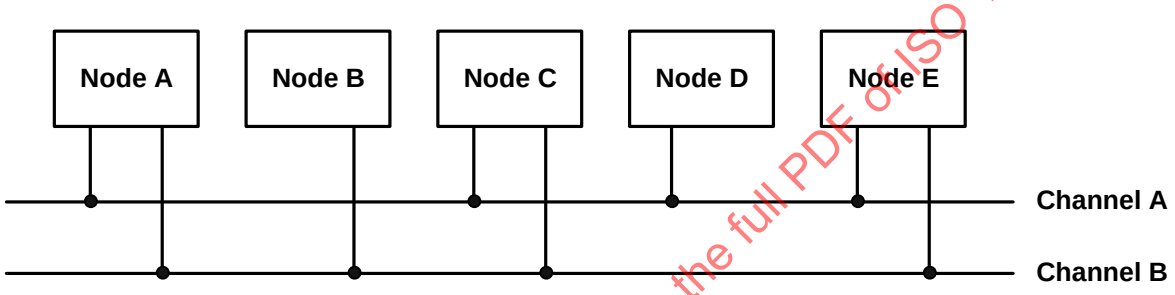



**Figure 6 — Dual channel bus configuration**

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

### 5.7.3 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 shall be free of closed rings, and there can be no more than two active stars on a network channel[10)].

NOTE there may be physical layer-related restrictions that limit the number of active stars for certain bit rates - see ISO 17458-4 for details.

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

The configuration of a single redundant star network is shown in Figure 7. The logical structure (i.e., the node connectivity) of this topology is identical with that shown in Figure 6. It is also possible to create a single, non-redundant star topology that has the same logical structure as the single bus mentioned above.

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

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

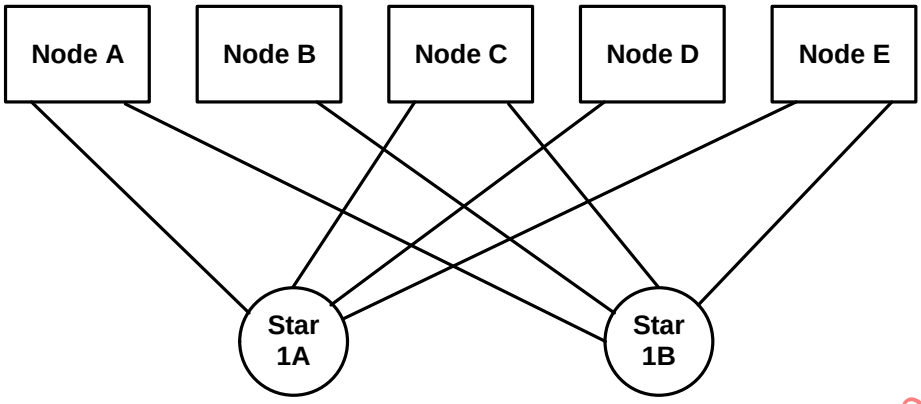



**Figure 7 — Dual channel single star configuration**

Figure 8 shows a single channel network built with two active stars. Each node has a point-to-point connection to one of the two active stars. The first active star (1A) is directly connected to the second active star (2A).

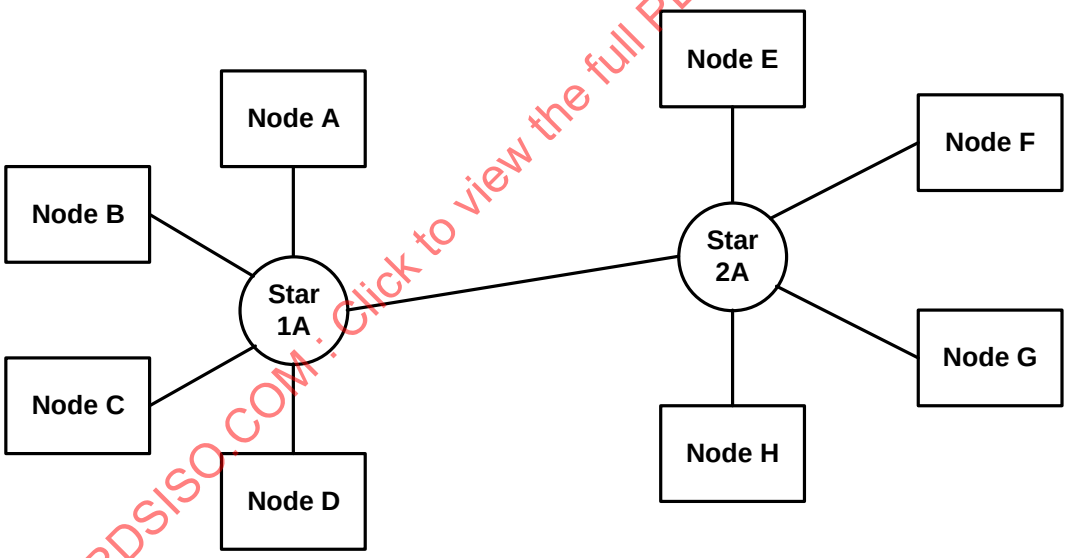



**Figure 8 — Single channel cascaded star configuration**



NOTE1 it is also possible to have a redundant channel configuration with cascaded stars. An example of such a configuration is Figure 9.

NOTE2 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.

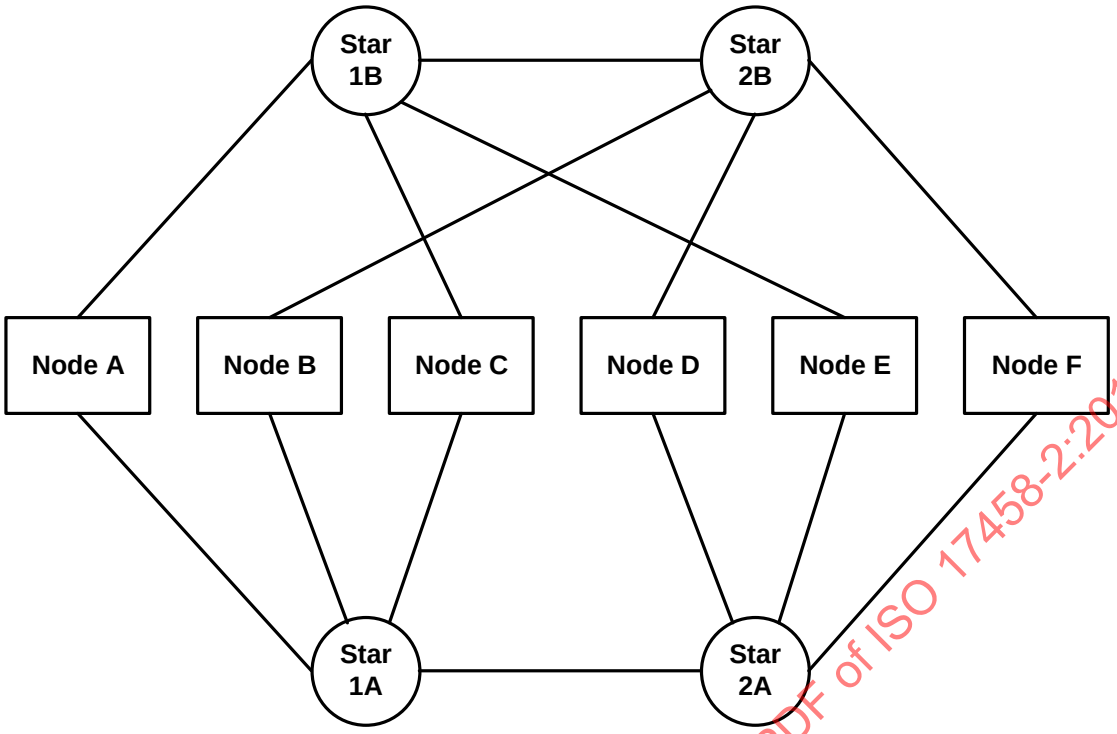



**Figure 9 — Dual channel cascaded star configuration**



### 5.7.4 Active star topology combined with a passive bus top

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

There are a large number of possible hybrid topologies, but only two representative topologies are shown here. Figure 10 shows an example of one type of hybrid topology. In this example, some nodes (nodes A, B, C, and D) are connected using point-to-point connections to an active star. Other nodes (nodes E, F, and G) are connected to each other using a bus topology. This bus is also connected to an active star, allowing nodes E, F, and G to communicate with the other nodes.

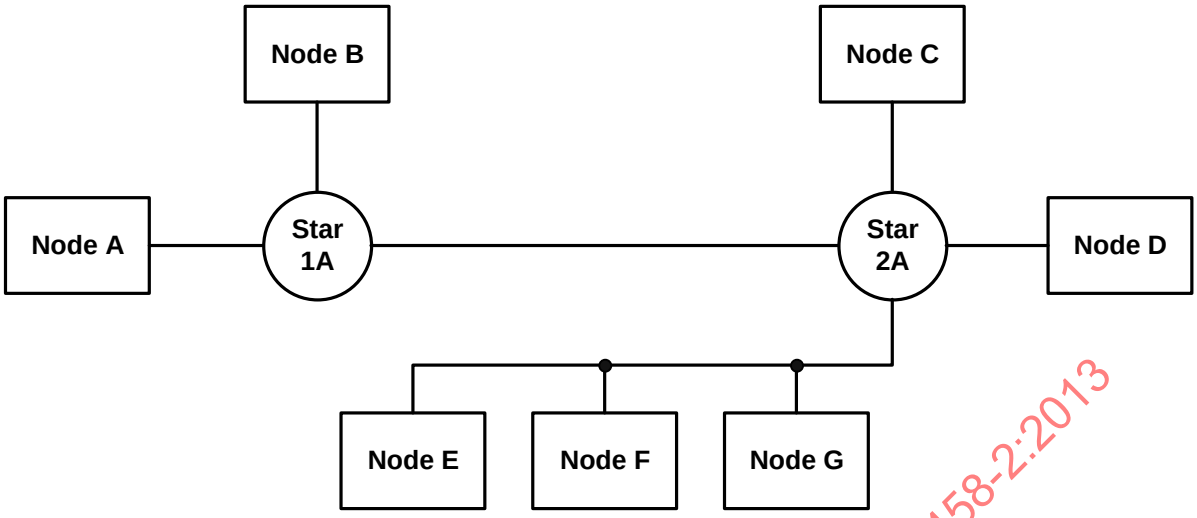



**Figure 10 — Single channel hybrid example**

A fundamentally different type of hybrid topology is shown in Figure 11. 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.

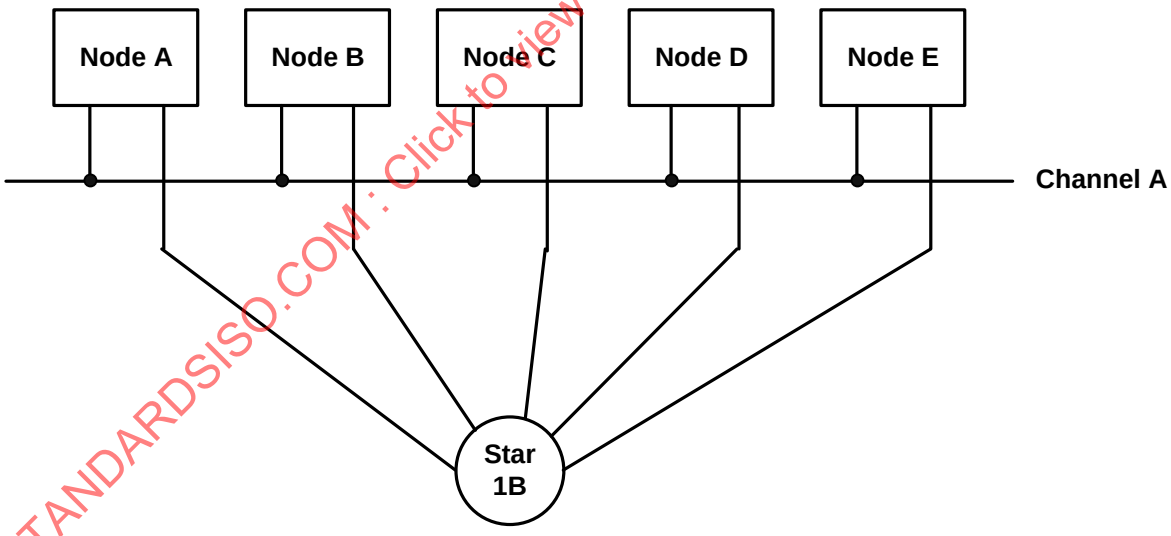



**Figure 11 — Dual channel hybrid example**

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

## 5.8 Example node architecture

### 5.8.1 Objective

This subclause 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 subclause is for reference only. The detailed specification of the interfaces is given in the electrical physical layer specification ISO 17458-4; references are made here to appropriate text passages from this document.

Note that an active star component can also function in a role similar to a bus driver via the use of the optional BD-CC interface as specified in ISO 17458-4. The following subclauses describe the node architecture under the assumption that the CC interfaces to the channel(s) via a bus driver rather than via an active star.

### 5.8.2 Overview

Figure 12 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.

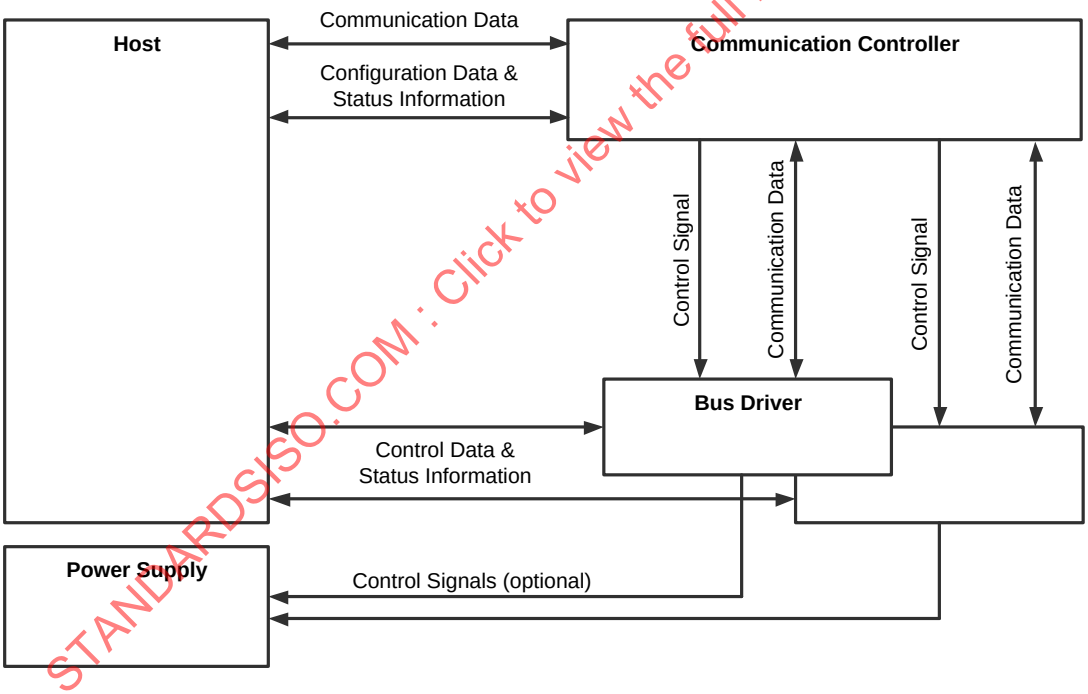





**Figure 12 — Logical interfaces**

### 5.8.3 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. Figure 13 illustrates the Host and the Communication Controller.

Details of the interface between the host and the communication controller are specified in clause 13.

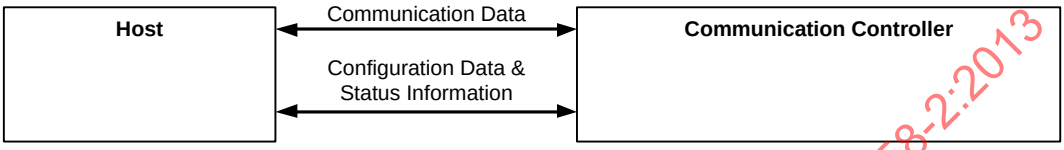



**Figure 13 — Host - communication controller interfaces**

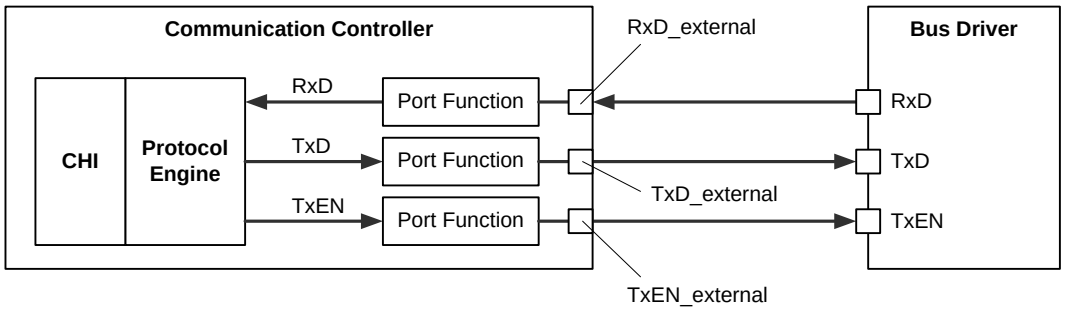
### 5.8.4 Communication controller - bus driver interface

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).

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

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

Figure 14 shows the connection between the communication controller and the bus driver and the internal connection between the protocol engine and the pins. This data link layer specification does not specify a device but the behaviour of the FlexRay protocol. In the following the data link layer specification only refers to the internal signals TxD, TxEN, and RxD as depicted in Figure 14.




**Key**

| | |
|---|---|
| CHI | Controller Host Interface |
| TxEN | Transmit enable |
| TxD | Transmit data |
| RxD | Receive data |

**Figure 14 — Communication controller - bus driver interface**

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

— I/O voltage level,

— ESD protection,

— behaviour during power up initialization, reset, or while depowered, and

— pin multiplexing (e.g. the connection of the external pins associated with the RxD_external, TxD_external and TxEN_external signals either to the FlexRay protocol engine (i.e., the RxD, TxD, and TxEN signals, respectively), to some other function inside the CC implementation[11)], or to nothing at all).

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

Some requirements (e.g. the electrical characteristics and timing) on the port functions and the TxD_external, TxEN_external and RxD_external signals are specified in ISO 17458-4.



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

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

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

### 5.8.5 Bus driver - host interface

#### 5.8.5.1 Overview

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 15) or by a Serial Peripheral Interface (SPI) (see option B in Figure 16).

#### 5.8.5.2 Hard wired signals (option A)

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

Figure 15 illustrates an example of a bus driver with a host interface option A.

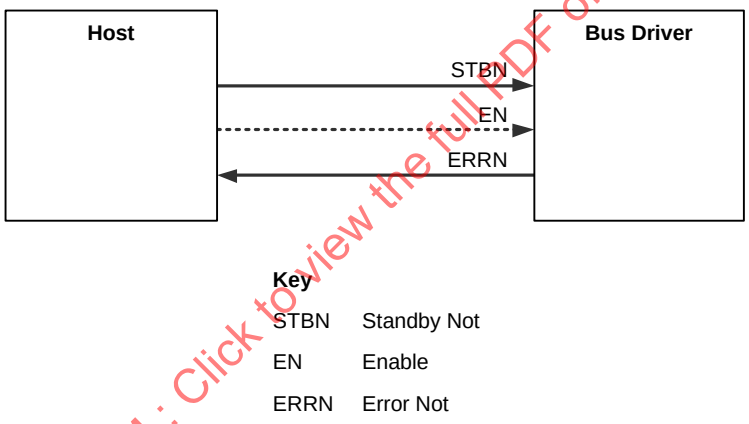



**Figure 15 — Example bus driver - host interface (option A)**

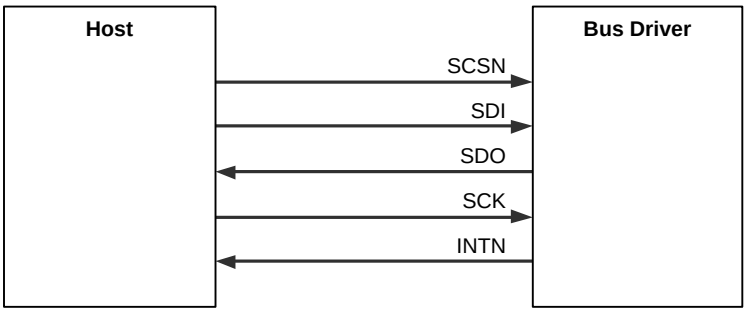
This interface is product specific; some restrictions are given in ISO 17458-4 that define minimum functionality to ensure interoperability.

#### 5.8.5.3 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).

Figure 16 illustrates an example of a bus driver with a host interface option B.

The electrical characteristics and timing of this interface are specified in ISO 17458-4.




**Key**

| | |
|---|---|
| SCSN | SPI chip select Not |
| SDI | SPI data input |
| SDO | SPI data output |
| SCK | SPI clock input |
| INTN | Interrupt |

**Figure 16 — Example bus driver - host interface (option B)**

### 5.8.6 Bus driver - power supply interface (optional)

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

Figure 17 illustrates an example of a bus driver with a power supply interface.


**Key**

| | |
|---|---|
| INH1 | Inhibit signal |

**Figure 17 — Bus driver - power supply interface**

The electrical characteristics and behaviour of the INH1 signal are specified in ISO 17458-4.

### 5.8.7 Time gateway interface

A time gateway sink node needs information on the schedule and clock synchronisation algorithm of its time gateway source node. The time gateway source node provides this information via the time gateway interface

to the time gateway sink node. This interface is unidirectional - no information flows back from the time gateway sink node to the time gateway source node. This interface is an optional feature only required to allow the node to be a time gateway source or time gateway sink node. Details of the interface between the time gateway source node and the time gateway sink node are specified in clause 12. The usage of this interface is indicated in the SDL diagrams by the EXP keyword on the transmitting end and the IMP keyword on the receiving end.

Figure 18 illustrates a time gateway interface.

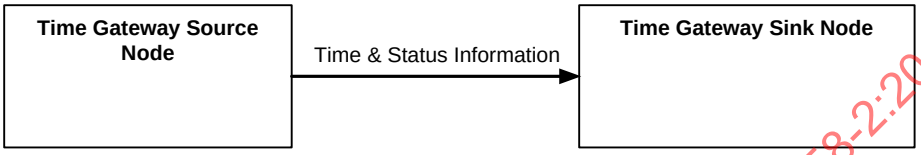



**Figure 18 — Time gateway interface**



#### 5.8.8 Testability requirements

ISO 17458-3 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.

## 6 Protocol operation control

### 6.1 Principles

#### 6.1.1 General

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

The primary protocol behaviour of FlexRay is embodied in four core mechanisms, each of which is described in a dedicated subclause of this specification for

— Coding and Decoding (see clause 7),

— Media Access Control (see clause 9),

— Frame and Symbol Processing (see clause 10), and

— Clock Synchronisation (see clause 12)

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 clause 13).

Each of the core mechanisms possesses modal behaviour that allows it to alter its fundamental operation in response to high-level mode changes of the node. Proper protocol behaviour 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 synchronisation of the core mechanisms is particularly evident during the wakeup, startup and reintegration procedures. These procedures are described in detail in clause 11. 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.

### 6.1.2 Communication controller power moding

Before the POC can perform its prescribed tasks the communication controller (CC) shall 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 19 depicts an overview of the CC power moding operation.

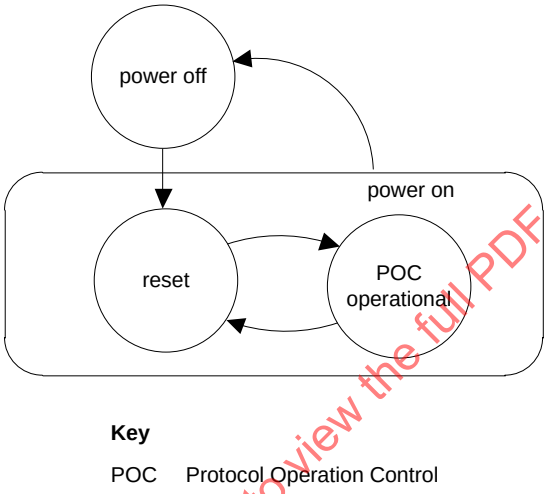



**Figure 19 — Power moding of the communication controller**

NOTE Figure 19 illustrates the state labelled *POC operational* which is not actually a specific state but rather a superset of all operational states of the protocol operation control (see Figure 21).

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



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

## 6.2 Description

### 6.2.1 Protocol operation control context

The relationships between the CHI, POC and the core mechanisms are depicted in Figure 20[15)].

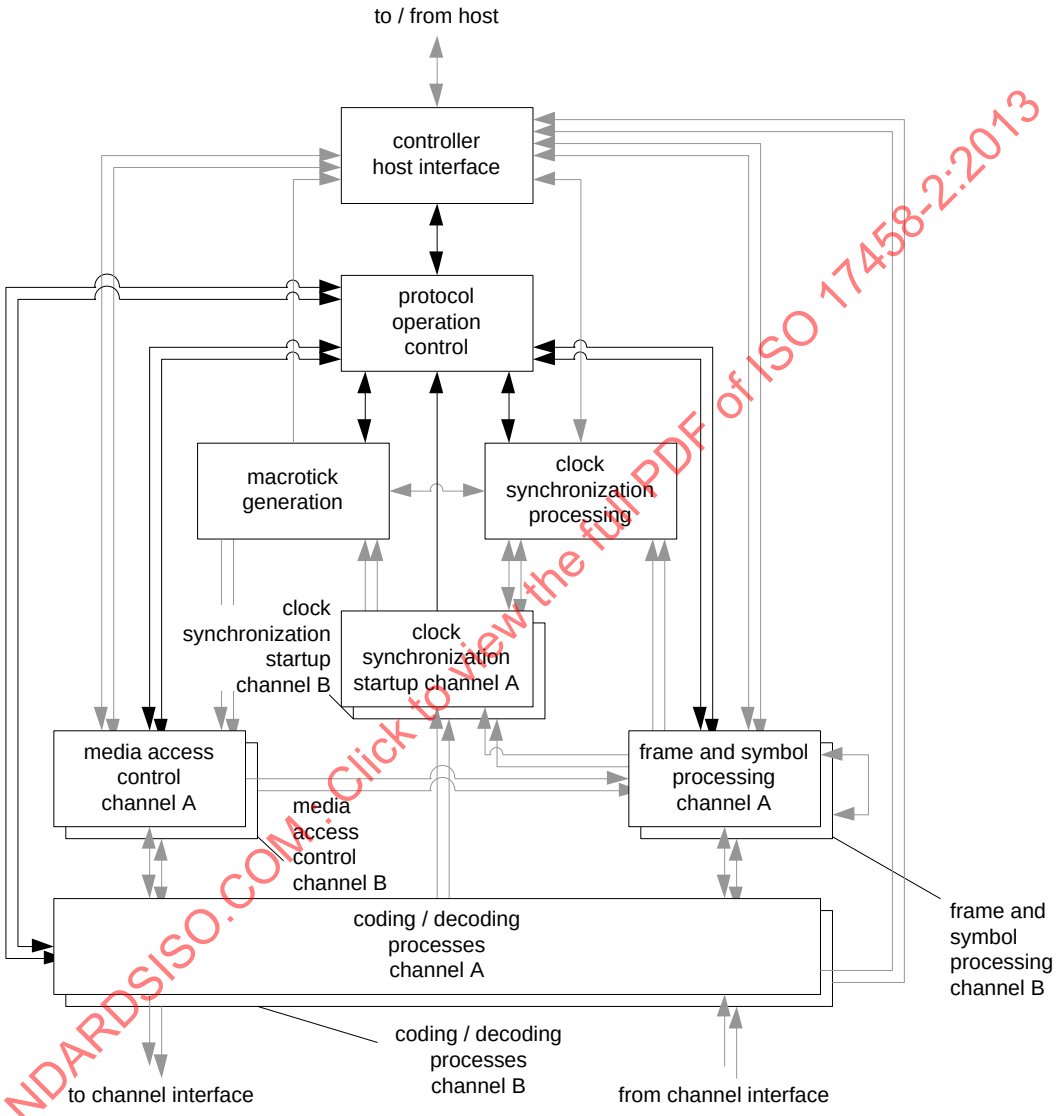





**Figure 20 — Protocol operation control context**

15) The dark lines represent data flows between mechanisms that are relevant to this subclause. The lighter gray lines are relevant to the protocol, but not to this clause.

### 6.2.2 Operational overview

#### 6.2.2.1 General

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 6.3.7) 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.

#### 6.2.2.2 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 clause 13).

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 behaviour corresponding to each command is captured in the SDL descriptions and accompanying text (see 6.3). They are briefly summarized in Table 4. If the host issues a specific CHI command while the POC is in a state other than the states shown in the "Where processed (POC States)" column in Table 4 the command shall be ignored (i.e., it shall have no effect on the protocol engine).

**Table 4 — CHI host command summary**

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

[a] The ALLOW_COLDSTART command is processed as described in Figure 31 except when the POC is in the *POC:integration listen* state, in which case it is processed by the SDL in Figure 150.

[b] In the *POC:coldstart collision resolution* state a deferred command is either processed at the end of the cycle or after a frame header or a CAS is received.

#### 6.2.2.3 Error conditions

##### 6.2.2.3.1 General

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.

##### 6.2.2.3.2 Errors causing immediate entry to the POC:halt state

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

— Product-specific error conditions such as BIST errors and sanity checks;

— Error conditions detected by the host that result in a FREEZE command being sent to the POC via the CHI;

— Fatal error conditions detected by the FSP process.

Product-specific errors are accommodated by the POC, but not described in this specification (see 6.3.4). Similarly, host detected error strategies are supported by the POC's ability to respond to a host FREEZE command (see 6.3.4), 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.

##### 6.2.2.3.3 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 synchronisation mechanism that are indicative of a problem, but that may not require immediate action due to the inherent fault tolerance of the clock synchronisation 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 synchronisation 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 synchronisation 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.

#### 6.2.2.4 POC status

In order for the host to react to POC state changes, the host shall 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.

**Definition: *T_POCStatus*** **(1)**

The basic POC status information is provided to the CHI using the *vPOC* data structure. *vPOC* is of type *T_POCStatus*.

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

**Definition: *T_POCState*** **(2)**

The *vPOC* structure is an aggregation of nine distinct status variables. *vPOC!State* is used to indicate the state of the POC and is based on the *T_POCState*.

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

*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 6.3.4). *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 6.3.5.1). *vPOC!CHIHaltRequest* is Boolean.

*vPOC!CHIReadyRequest* is used to indicate that a request has been received from the CHI to enter the *POC:ready* state at the end of the communication cycle (see 6.3.5.1). *vPOC!CHIReadyRequest* is Boolean.

*vPOC!ColdstartNoise* indicates noisy channel conditions during *POC:coldstart listen* if the coldstart attempt of a leading coldstart node was completed successfully (see 11.3). *vPOC!ColdstartNoise* is Boolean.

**Definition: *T_SlotMode*** **(3)**

*vPOC!SlotMode* is used to indicate what slot mode the POC is in (see 6.3.5.2, 6.3.8.2.3, and 6.3.8.2.4). *vPOC!SlotMode* is based on the *T_SlotMode* formal definition.

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

```
endnewtype;
```

**Definition: *vPOC!ErrorMode*** **(4)**

*vPOC!ErrorMode* is used to indicate what error mode the POC is in (see subclauses 6.3.8.2.3 and 6.3.8.2.4). *vPOC!ErrorMode* is based on the *T_ErrorMode* formal definition.

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

**Definition: *vPOC!WakeupStatus*** **(5)**

*vPOC!WakeupStatus* is used to indicate the outcome of the execution of the WAKEUP mechanism (see Figure 30 and 11.2.3.2). *vPOC!WakeupStatus* is based on the *T_WakeupStatus* formal definition.

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

The meaning of the individual *T_WakeupStatus* values is outlined in 11.2.3.2.

**Definition: *vPOC!StartupState*** **(6)**

*vPOC!StartupState* is used to indicate the current substate of the startup procedure (see 11.3.5). *vPOC!StartupState* is based on the *T_StartupState* formal definition.

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

The individual *T_StartupState* values are the states within the STARTUP mechanism in 11.3.5.

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 6.3.8.2.5.



### 6.2.2.5 SDL considerations for single channel nodes

FlexRay supports configurations where a node is only attached to one of the two possible FlexRay channels (see 5.7).

Process instantiation is depicted in Figure 23. The channel specific processes are readily identifiable by the "_A" or "_B" text in the process names. The POC only instantiates the channel specific processes related to channels that are actually attached.

Process termination signal generation is also depicted in Figure 23. The channel specific signals are identifiable by the "_A" or "_B" text in the signal names. Termination signals sent to a channel specific process that is not instantiated will have no effect.

Process moding signals are generated throughout the POC. Figure 22 is an example that includes all of these moding signals. The channel specific moding signals are identifiable by the "on A" or "on B" text in the signal names. Moding signals, or any other signals, sent to a channel specific process that is not instantiated will have no effect.

## 6.3 The protocol operation control process

### 6.3.1 General

This subclause contains the formalized specification of the POC process. Figure 21 depicts an overview of the POC states and how they interrelate[16].

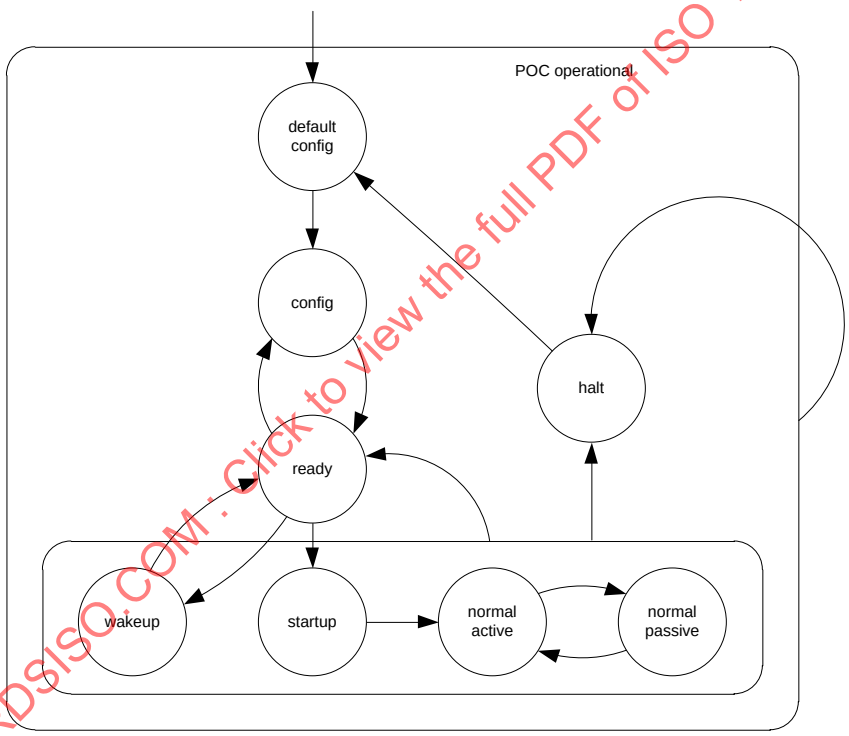



**Figure 21 — Overview of protocol operation control**



### 6.3.2 POC SDL utilities

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

In the SDL that follows, there are several instances where the POC transitions to the *POC:ready* or *POC:halt* states. Prior to doing so, the core mechanisms have to be moded appropriately. The two macros in Figure 22 perform these tasks. PROTOCOL_ENGINE_READY modes the core mechanisms appropriately for entry to

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

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

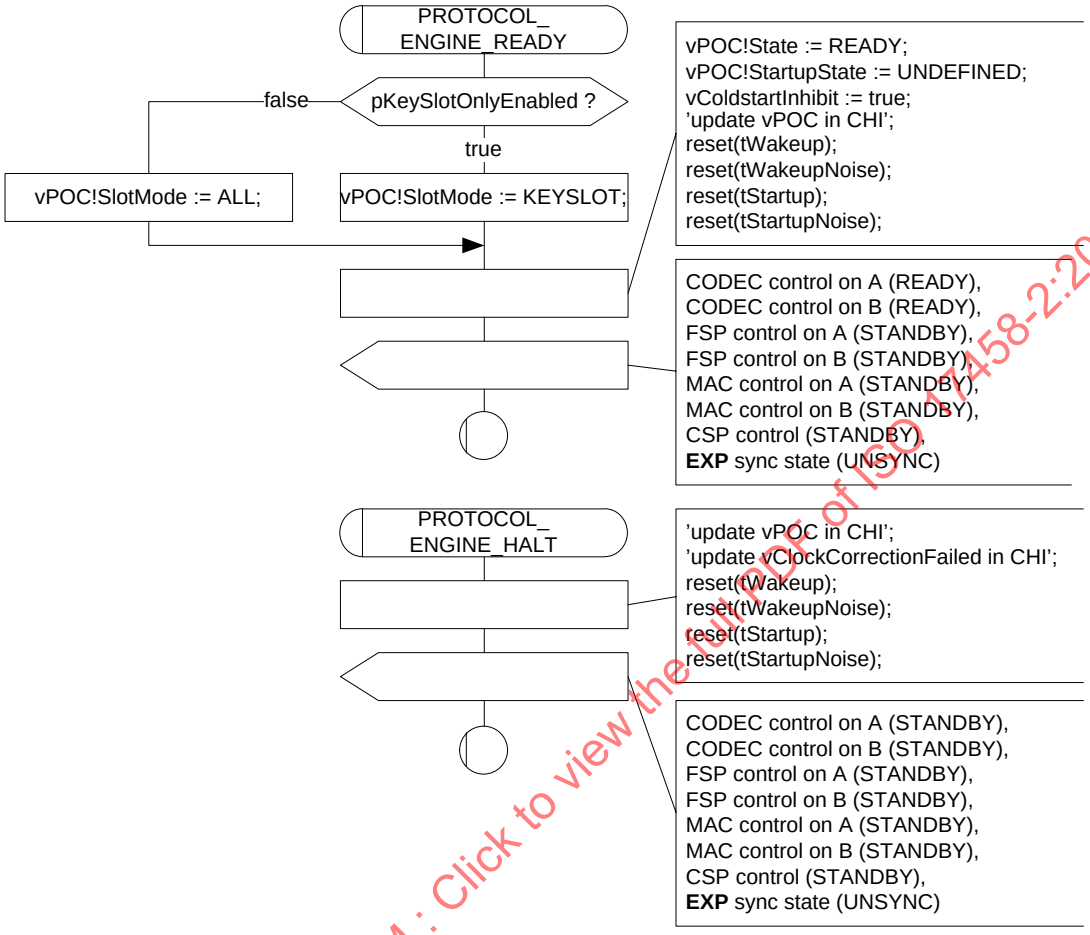



**Figure 22 — Macros to mode the core mechanisms for transitions to the POC:ready and POC:halt states [POC]**

The SDL processes associated with the core mechanisms are created simultaneously by the POC. While the processes shall terminate themselves, the POC is also responsible for simultaneously triggering this in all of the processes. Figure 23 depicts the macros for performing these two tasks.

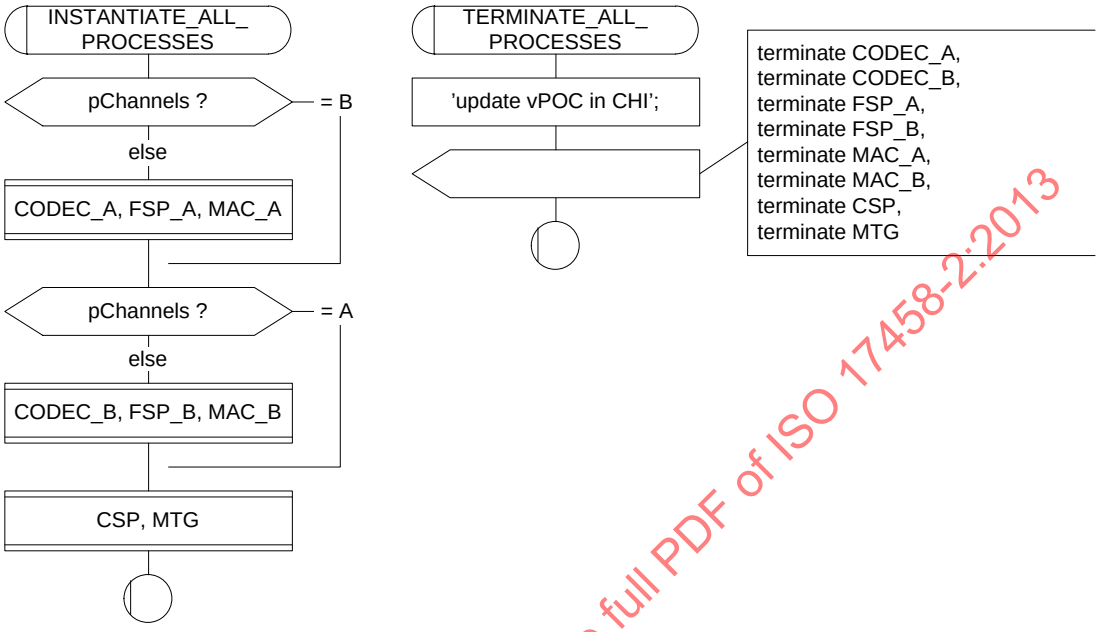



**Figure 23 — Macros for creating and terminating processes [POC]**



### 6.3.3 SDL organization

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

— behaviours corresponding to host commands that preempt the regular behavioural flow;

— behaviour that brings the POC to the *POC:ready* state;

— behaviour leading from the *POC:ready* state to the *POC:normal active* state;

— behaviour once the *POC:normal active* state has been reached, i.e., during "normal operation".

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

### 6.3.4 Preempting commands

There are two commands (FREEZE and IMMEDIATE_READY) that are used to preempt the normal behavioural flow of the POC. They are depicted in Figure 24. It should be emphasized that these commands also apply to the behaviour contained in the Wakeup and Startup macros (see Figure 30) that is detailed in clause 11.

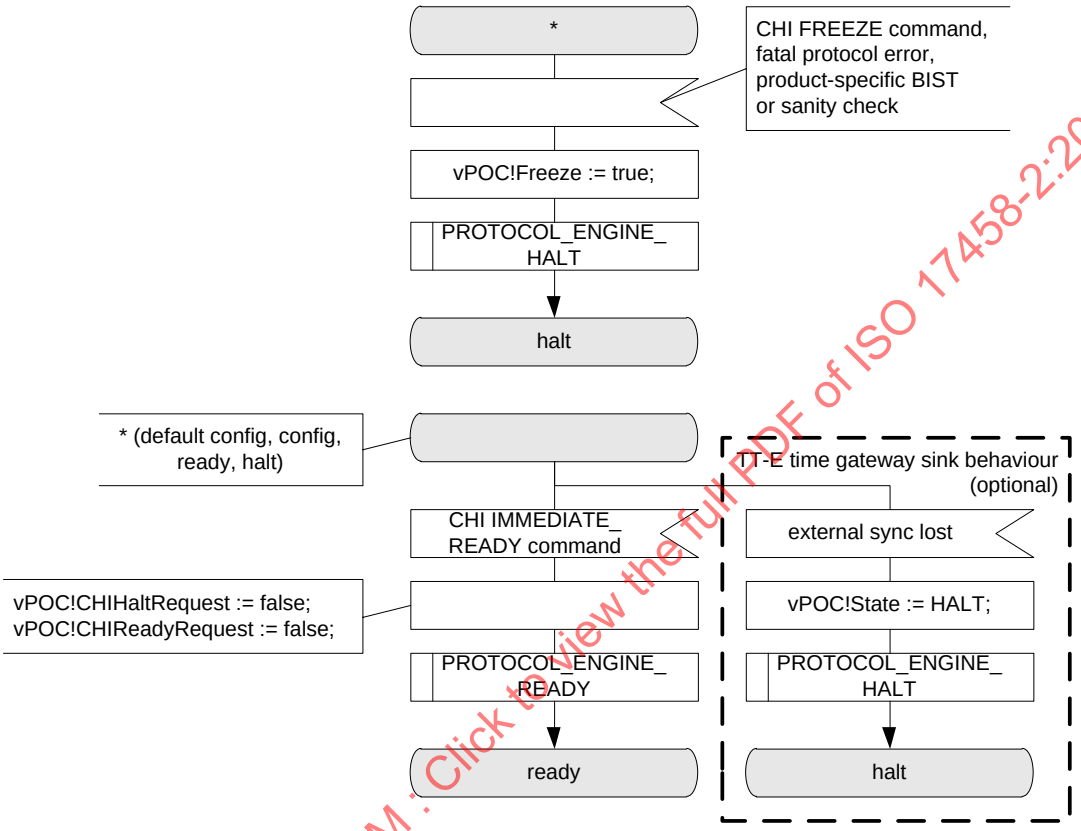



**Figure 24 — POC preempting immediate commands [POC]**

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

— a host FREEZE command relayed to the POC by the CHI;

— a *fatal protocol error* signalled by the FSP process;

— a product-specific error detected by a built-in self-test (BIST) or sanity check;

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

17) 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.

At the host's discretion, the ongoing operation of the POC can be interrupted by immediately placing the POC in the *POC:ready* state. In response to this command, the POC modes the core mechanisms appropriately (see 6.3.2), and then transitions to *POC:ready*.

### 6.3.5 Deferred commands

#### 6.3.5.1 DEFERRED_HALT, DEFERRED_READY and CLEAR_DEFERRED commands

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

The CHI may relay the DEFERRED_HALT command from the host at any time the POC is in a state other than *POC:halt*. The CHI may relay the DEFERRED_READY command from the host at any time that it would be allowed to relay the IMMEDIATE_READY command (see Table 4). If no communication is ongoing the effects of these commands is immediate. If communication is ongoing the effects of the commands are deferred, in most cases to the processing at the end of the cycle. If the POC is in the *POC:wakeup send* state a deferred command is processed after transmission of a complete WUP or detection of a wakeup collision. If the POC is in the *POC:coldstart collision resolution* state a deferred command is processed at the end of the cycle or after a frame header or a CAS is received. In all cases where the processing of the commands is deferred it is necessary to capture indications that the commands have occurred so that processing can take place at the appropriate time. Figure 25 depicts the procedure that captures these commands.

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

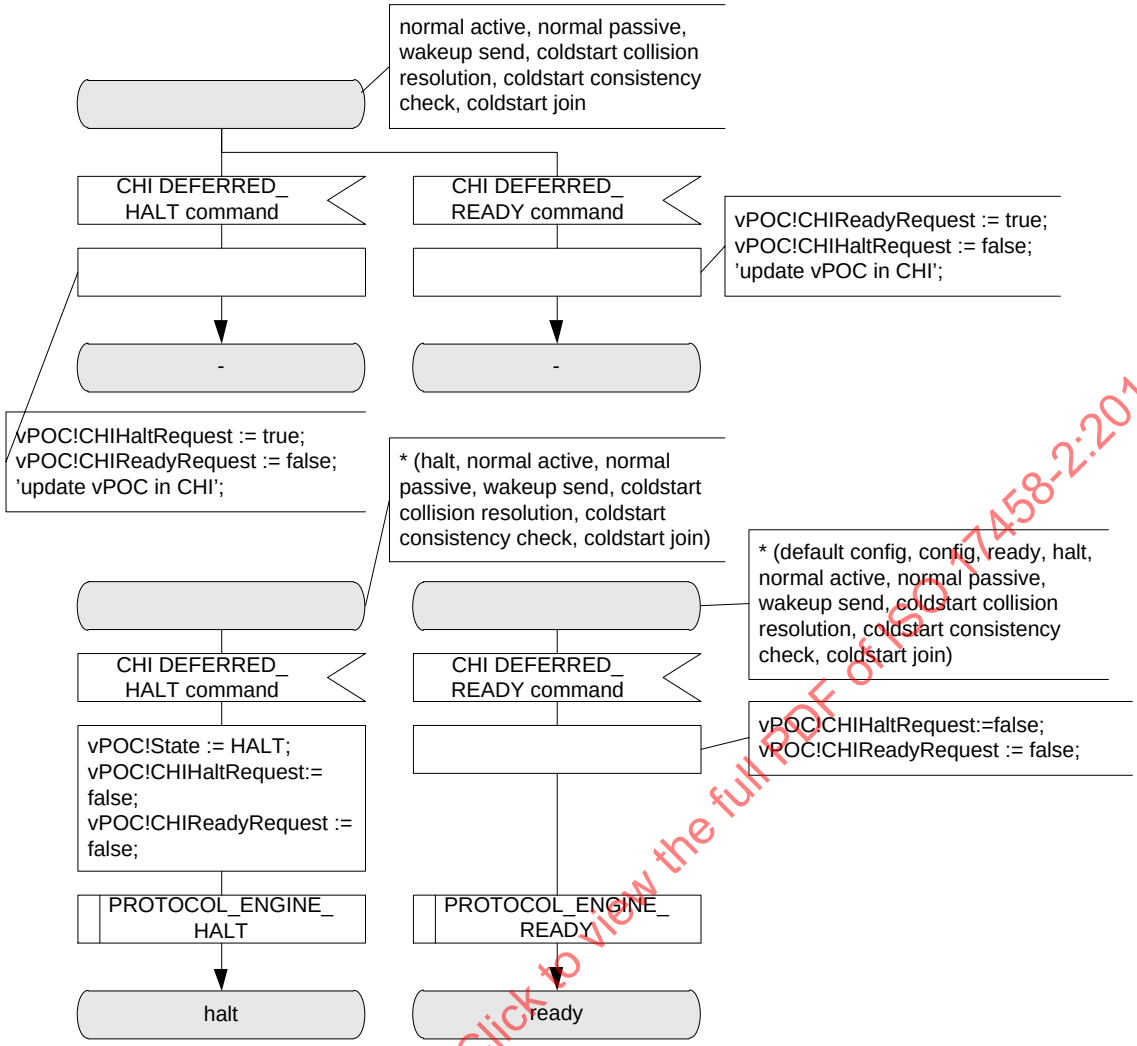



**Figure 25 — POC preempting deferred commands [POC]**



The DEFERRED_HALT command shall be captured by setting the *vPOC!CHIHaltRequest* value to true if the command is not immediately processed. When processed at the end of the current cycle, the DEFERRED_HALT command will cause the POC to enter the *POC:halt* state. This is the standard method used by the host to shutdown the CC.

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

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

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

Figure 26 depicts the HANDLE_DEFERRED_CHI_COMMANDS macro which is used in clause 11.

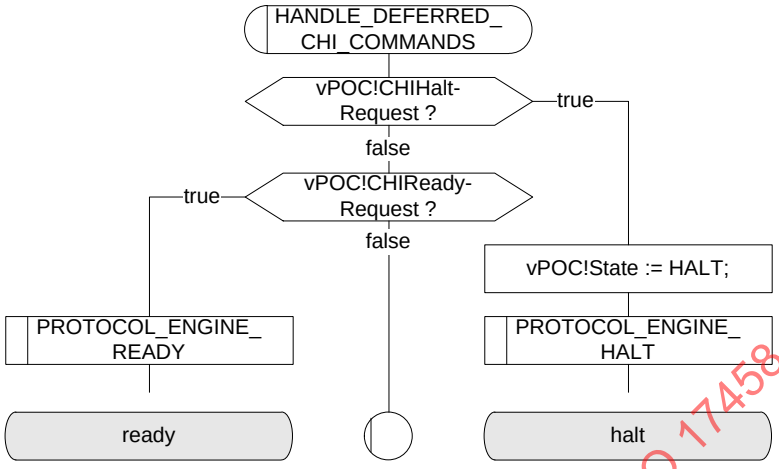



**Figure 26 — Macro to handle deferred CHI commands [POC]**

NOTE The macro shown in Figure 26 contains transitions to states that are defined outside of the macro (*POC:ready* and *POC:halt*). The reader should take care when interpreting this macro in higher level SDL diagrams, as the exits to other states will not appear in the higher-level diagram.

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

The CLEAR_DEFERRED command shall set the *vPOC!CHIReadyRequest* value and the *vPOC!CHIHaltRequest* value immediately to false.

NOTE The CLEAR_DEFERRED command is not able to delete a deferred command in all POC states because in a number of POC states a deferred command is executed immediately (see Table 4).

Figure 27 illustrates the cancelation of deferred commands.

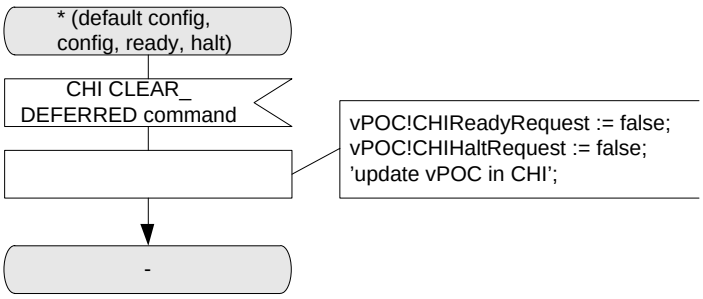



**Figure 27 — Cancelation of deferred commands [POC]**

#### 6.3.5.2 ALL_SLOTS command

The CHI may relay the ALL_SLOTS command from the host at any time while the POC is in the *POC:normal active* or *POC:normal passive* states. Its effect is realized during the processing at the end of the cycle, but it is necessary to capture an indication that the command has occurred so that appropriate processing will occur at cycle end. Figure 28 depicts the procedure that captures this command.

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

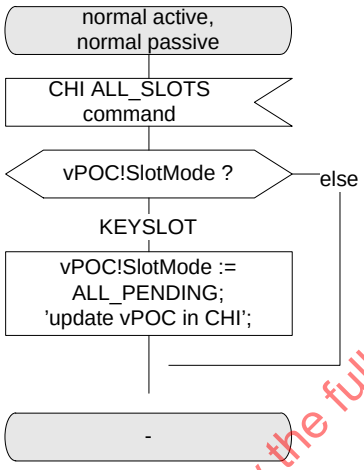



**Figure 28 — Capture of the ALL_SLOTS command for end-of-cycle processing [POC]**

### 6.3.6 Reaching the POC:ready state

#### 6.3.6.1 State sequence to reach the POC:ready state

The tasks that the POC executes in order to reach the *POC:ready* state serve primarily as an initialisation process for the POC and the core mechanisms. This initialisation process is depicted in Figure 29.

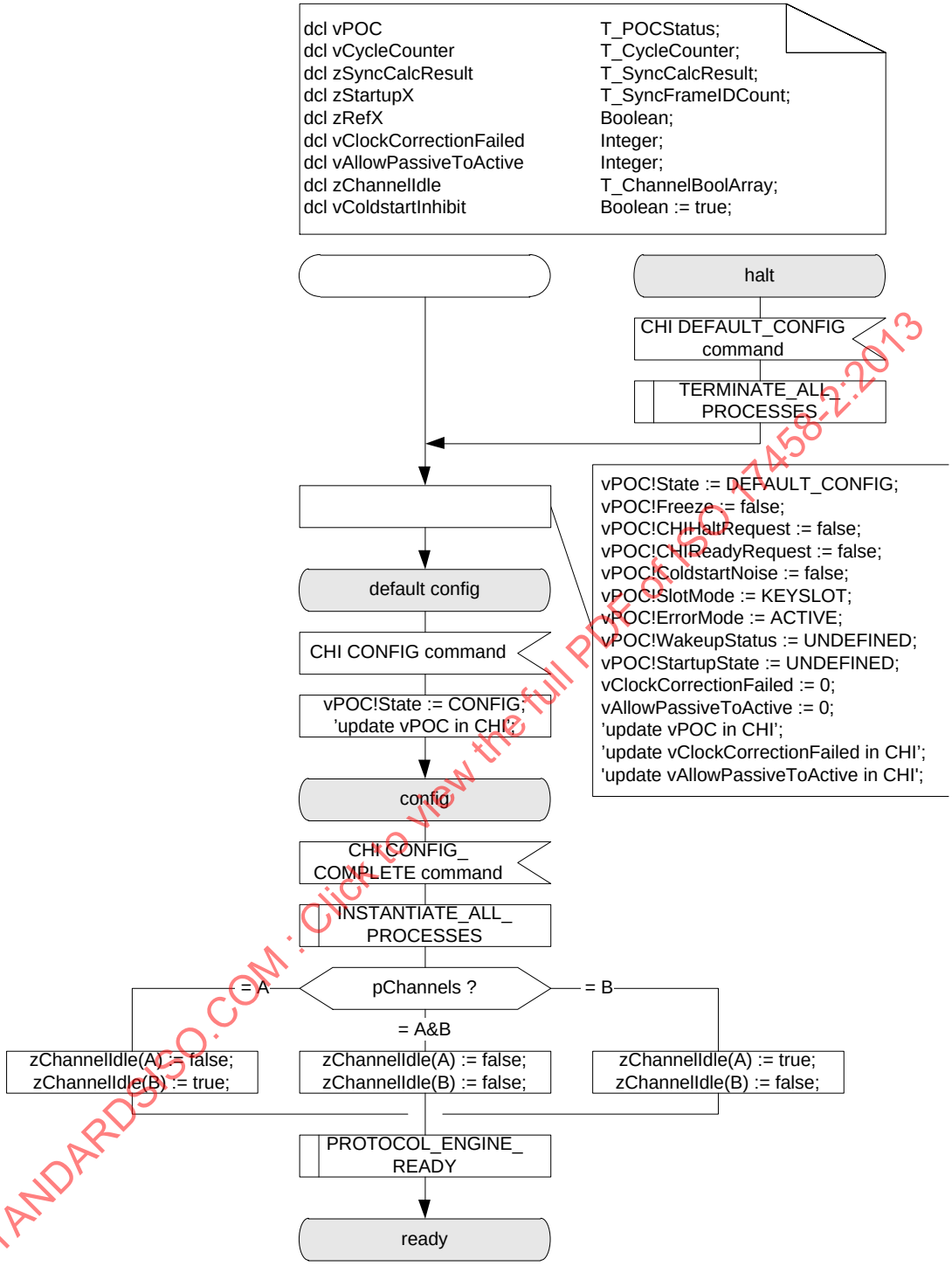





**Figure 29 — Reaching the POC:ready state [POC]**

The POC shall enter the *POC:default config* state when the CC enters the *POC operational* state (see 6.1.2). 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 initialisation process.

Prior to entering the *POC:default config* state the POC shall initialize the elements of the *vPOC* data structure that are used to communicate the POC status to the CHI. With the exception of *vPOC!SlotMode*, the values

assumed by the *vPOC* elements are obvious initial values and are depicted in Figure 29. The initial value of *vPOC!SlotMode* is defaulted to KEYSLOT until the configuration process is carried out to set it to the value desired by the host.

In the *POC:default config* state the POC awaits the explicit command from the host to enable configuration. The POC shall enter the *POC:config* state in response to the CONFIG command. Configuration of the CC is only allowed in the *POC:config* state and this state can only be entered with an explicit CONFIG command issued while the POC is in the *POC:default config* state or the *POC:ready state* (see 6.3.7).

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

The POC shall transition to the *POC:ready* state in response to the CONFIG_COMPLETE command. On this transition, the POC shall create all of the core mechanism processes, incorporating the configuration values that were set in the *POC:config* state. It shall then update *vPOC* to reflect the new state, the newly configured value of slot mode [18)] and the initial value of *vColdstartInhibit*. It shall then command all of the core mechanisms to their appropriate mode (see 6.3.2). The POC then transitions to the *POC:ready* state.

#### 6.3.6.2 Default configuration requirements

*POC:default config* is a state that ensures that the CC has a defined, stable default configuration prior to application-specific configuration that takes place in the *POC:config* state. Upon entry into the POC:default config state the CC shall ensure that all configuration data or control data described in subclauses 13.3.1.1, 13.3.1.2.2, 13.3.2.6.2, and 13.3.2.11 are set to defined values as described below.

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

— all buffers (including FIFO buffers) shall be configured such that they can neither transmit nor receive;

— no slot shall be assigned for transmission or reception;

— the payload data valid flag of all message buffers shall be set to false;

— *pKeySlotID* and *pSecondKeySlotID* shall be set to 0;

— *pKeySlotUsedForStartup* shall be set to false;

— *pTwoKeySlotMode* shall be set to false;

— *pWakeupPattern* shall be set to 0;

— *pExternalSync* (if applicable) shall be set to false;

— no transmissions of WUDOP's or MTS's are scheduled (see 13.3.1.2.2).

Other than the specific case described below, all other configuration data defined in subclauses 13.3.1.1, 13.3.2.6.2, and 13.3.2.11 shall be set to implementation dependent predefined initialisation values. The initialisation values for each individual configuration in the default configuration shall be described in the documentation of the implementation.

---

18) The value is determined by the node configuration, *pKeySlotOnlyEnabled*, a Boolean used to indicate whether the key slot only mode is enabled. This supports an optional strategy to limit frame transmissions following startup to the key slots until the host confirms that the node is synchronized to the cluster and enables the remaining transmissions with an ALL_SLOTS command (see 6.3.5.2).

An exception to the previous requirement is the configuration for *pChannels* (refer to 13.3.1.1.3). For this parameter there is no requirement for a transition into the *POC:default config* state to cause this configuration parameter to be set to any specific value (it is allowed, but not required).

### 6.3.7 Reaching the POC:normal active state

#### 6.3.7.1 Host commands before reaching the POC:normal active state

Following the initialisation sequence (see 6.3.6) the CC resides in the *POC:ready* state (see Figure 30). From this state the CC is able to perform the necessary tasks to start or join an actively communicating cluster. There are three POC actions that can take place, each of which is initiated by a specific command from the CHI. These commands are WAKEUP, RUN, and CONFIG.

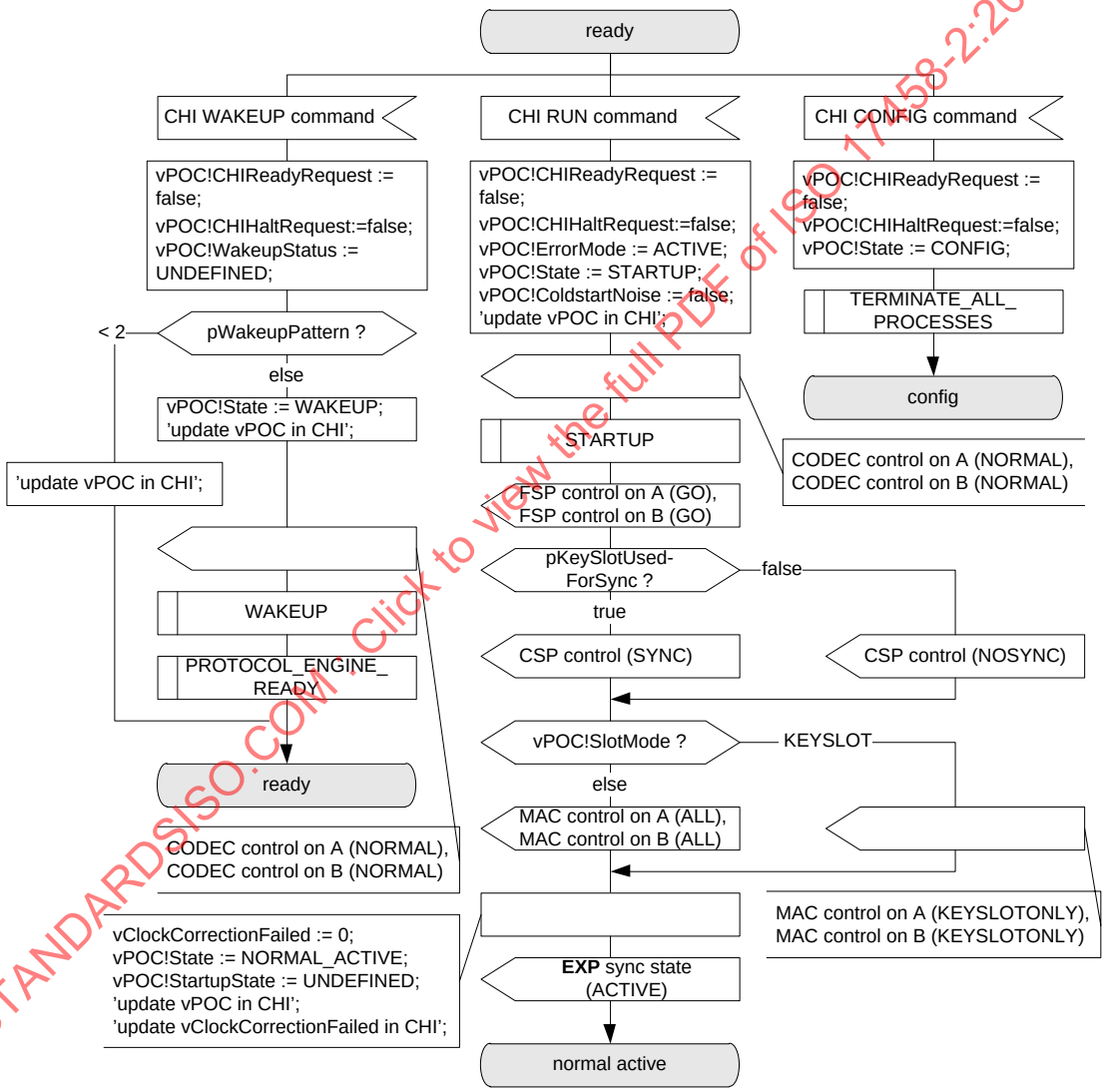



**Figure 30 — POC behaviour in preparation for normal operation [POC]**

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 6.3.2), 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 11.2, and is represented in Figure 30 by the WAKEUP macro invocation. On completion of the wakeup procedure, the POC shall mode all the core mechanisms appropriately for *POC:ready* (see 6.3.2) and return to the *POC:ready* state.

The RUN command shall cause the POC to commence a sequence of tasks to bring the POC to normal operation, i.e. the *POC:normal active* state. First, all internal status variables are reset to their starting values[19]. Then the startup procedure is executed. In Figure 30 this is represented by the STARTUP macro invocation. This procedure is described in detail in 11.3. This procedure modes the core mechanisms appropriately to perform the sequence of tasks necessary for the node to start or enter an actively communicating cluster.

The startup procedure results in the node being synchronized to the timing of the cluster. At the end of the communication cycle, the POC shall mode the core mechanisms depending on the values of *vPOC!SlotMode* and the configuration *pKeySlotUsedForSync* as depicted in Figure 30:

— the FSP mechanism shall be moded to GO for both channels;

— if the node is a sync node (*pKeySlotUsedForSync* is true) CSP shall be moded to SYNC mode. Otherwise, CSP shall be moded to NOSYNC;

— if the node is currently in key slot only mode (*vPOC!SlotMode* is KEYSLOT), then the POC shall mode the MAC to KEYSLOTONLY mode on both channels. If the node is not currently in key slot only mode (*vPOC!SlotMode* is ALL), then the POC shall mode the MAC to ALL mode on both channels.

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

#### 6.3.7.2 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 behaviour captured in those macros is documented in clause 11 and is largely self-contained. However, there are two exceptions and they are depicted in Figure 31.



19) This is necessary because the *POC:ready* state may have been entered due to a DEFERRED_READY or IMMEDIATE_READY command from the CHI that caused the POC to enter *POC:ready* from a state where the status variables had already been altered (see 6.3.3).

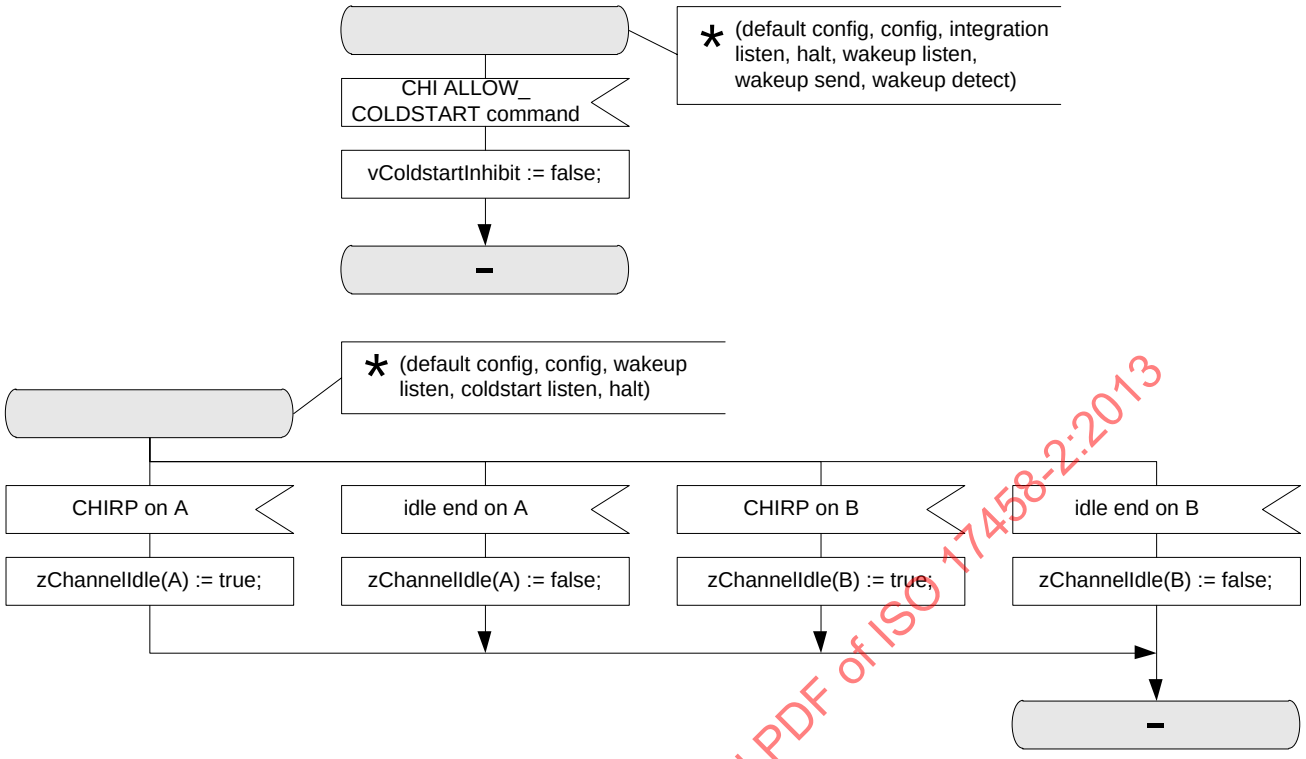



**Figure 31 — Conditions detected in support of the wakeup and startup procedures [POC]**



The behaviour of the POC during startup is influenced by whether the node is currently inhibited from acting as a leading coldstart node in the startup process (see 11.3.4). A restriction on the node's ability to act as a leading coldstart node is reflected in the Boolean variable *vColdstartInhibit*. While this value is acted upon in the startup procedure, the CHI also allows the host to change the variable to false by issuing the ALLOW_COLDSTART command while in the *POC:ready* state and any of the states that are part of startup, normal active, or normal passive.

The POC sets the value of the *vColdstartInhibit* variable to true on all transitions into the *POC:ready* state. A system designer shall be aware of this behaviour, and shall ensure that the ALLOW_COLDSTART commands are issued such that *vColdstartInhibit* has the desired value when the RUN command is issued[20)].

In a similar manner, both the wakeup and startup procedures shall 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 31 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 signalled for the corresponding channel by the BITSTRB processes (see 7.4.2).

Similarly, the POC shall change the value of the appropriate *zChannelIdle* array element to true whenever a channel idle recognition point (CHIRP) is signalled for the corresponding channel by the BITSTRB processes (see 7.4.2).

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

The *zChannelIdle* array is of type *T_ChannelBoolArray*.

**Definition: *T_ChannelBoolArray*** **(7)**

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

The index to the array is the channel identifier, which is of type T_Channel.

**Definition: *T_Channel*** **(8)**

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

### 6.3.8 Behaviour during normal operation

#### 6.3.8.1 General

Other than the commands that preempt regular behavioural flow (see 6.3.4), there are two components of the POC behaviour once normal operation has begun.

— The capture of deferred host commands that the CHI relays to the POC for later processing (see 6.3.5).

— The cyclical processing of error status information and deferred host commands at the end of each cycle.

#### 6.3.8.2 Cyclic behaviour

##### 6.3.8.2.1 Recurring Tasks

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

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

#### 6.3.8.2.2 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 synchronisation 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 32.

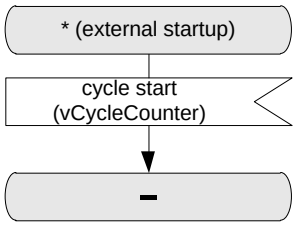



**Figure 32 — POC determination of the cycle counter value [POC]**

#### 6.3.8.2.3 POC:normal active state

Following a successful startup the POC will reside in the *POC:normal active* state (see 6.3.7). As depicted in Figure 33 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 157) signals the completion of the clock correction calculation to the POC using the *SyncCalcResult* signal.

This signal results in the following.

— 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 6.3.5.2).

— The POC then performs a sequence of error checking tasks whose outcome determines the subsequent behaviour. This task sequence is represented by the invocation of the NORMAL_ERROR_CHECK macro in Figure 33. The details of this task sequence are described in 6.3.8.2.5.2. As a result the POC will be in one of the following states.

  — If the *vPOC!ErrorMode* is ACTIVE and the CHI did not relay a DEFERRED_HALT or a DEFERRED_READY command to the POC in the preceding communication cycle (see 6.3.5.1), the POC shall remain in the *POC:normal active* state.

  — If the *vPOC!ErrorMode* is PASSIVE and the CHI did not relay a DEFERRED_HALT or a DEFERRED_READY command to the POC in the preceding communication cycle (see 6.3.5.1), the POC shall mode the MAC and CSP to halt frame transmission and transition to the *POC:normal passive* state.

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

  — If the CHI did relay a DEFERRED_HALT command to the POC in the preceding communication cycle (see 6.3.5.1), the POC shall stop the execution of the core mechanisms by moding them to STANDBY and transition to the *POC:halt* state.

— If the CHI did relay a DEFERRED_READY command to the POC in the preceding communication cycle (see 6.3.5.1), the POC shall stop the execution of the core mechanisms by moding them to STANDBY and transition to the *POC:ready* state.

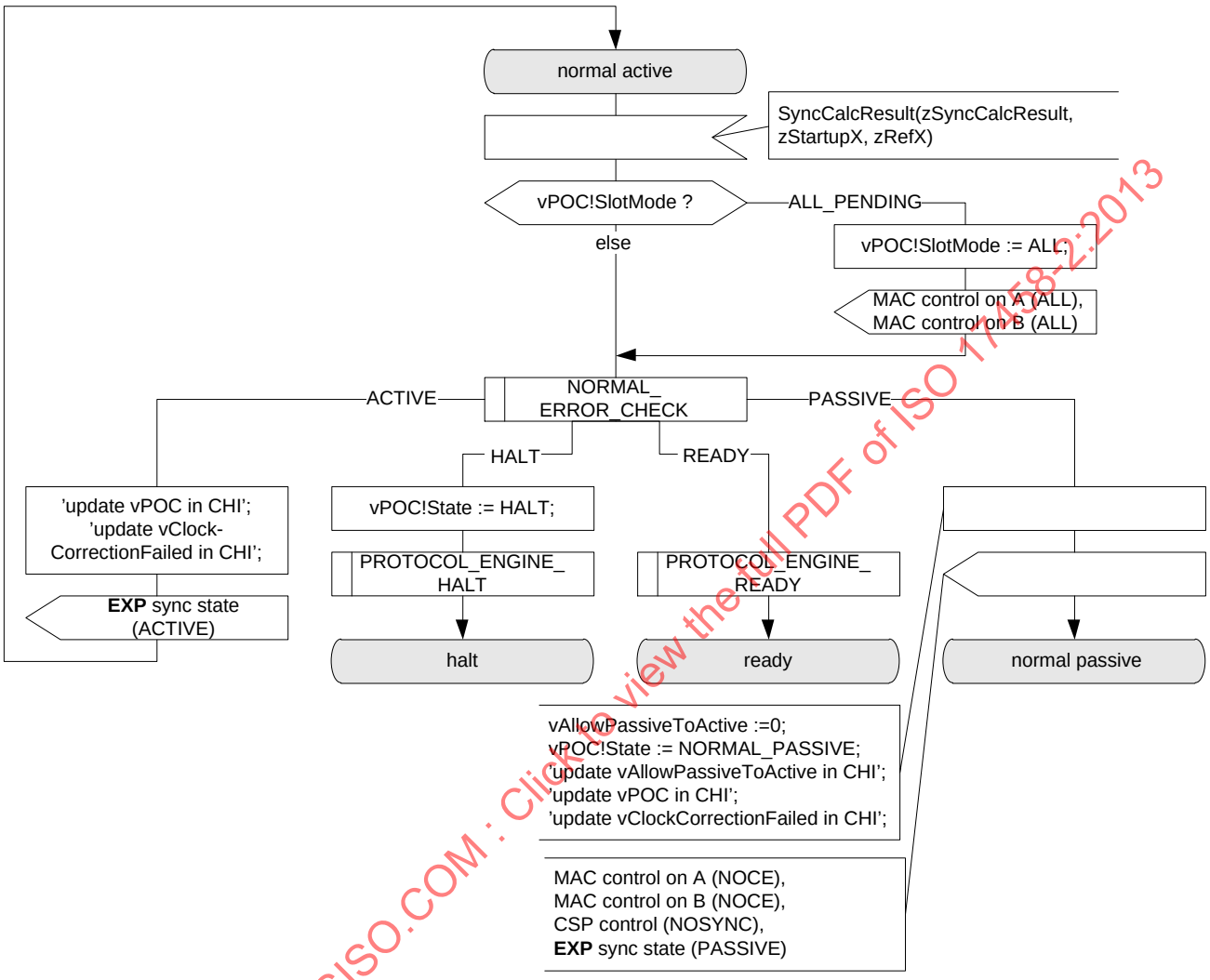



**Figure 33 — Cyclical behaviour in the POC:normal active state [POC][21)]**



#### 6.3.8.2.4 POC:normal passive state

The POC's behaviour in the *POC:normal passive* state is analogous its behaviour in the *POC:normal active* state (see 6.3.8.2.3). As depicted in Figure 34 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 157) signals the completion of the clock correction calculation to the POC using the *SyncCalcResult* signal.

This signal results in the following.

21) *zStartupX* is *zStartupNodes* in even cycles and *zRxStartupPairs* in odd cycles. *zRefX* is *zRefNode* in even cycles and *zRefPair* in odd cycles. See Figure 157 for details.

— 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 6.3.5.2).

— The POC then performs a sequence of error checking tasks whose outcome determines the subsequent behaviour. This task sequence is represented by the invocation of the PASSIVE_ERROR_CHECK macro in Figure 34. The details of this task sequence are described in 6.3.8.2.5.3. As a result the POC will be in one of the following states.

  — If the *vPOC!ErrorMode* is ACTIVE and the CHI did not relay a DEFERRED_HALT or a DEFERRED_READY command to the POC in the preceding communication cycle (see 6.3.5.1), the POC shall mode the MAC and CSP mechanisms to support resumption of frame transmission based on whether the node is a sync node (*pKeySlotUsedForSync* is true) and whether the node is currently in key slot only mode, and then transition to the *POC:normal active* state.

  — If the *vPOC!ErrorMode* is PASSIVE and the CHI did not relay a DEFERRED_HALT or a DEFERRED_READY command to the POC in the preceding communication cycle (see 6.3.5.1), the POC shall remain in to the *POC:normal passive* state.

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

  — If the CHI did relay a DEFERRED_HALT command to the POC in the preceding communication cycle (see 6.3.5.1), the POC shall stop the execution of the core mechanisms by moding them to STANDBY and transition to the *POC:halt* state.

  — If the CHI did relay a DEFERRED_READY command to the POC in the preceding communication cycle (see 6.3.5.1), the POC shall stop the execution of the core mechanisms by moding them to STANDBY and transition to the *POC:ready* state.

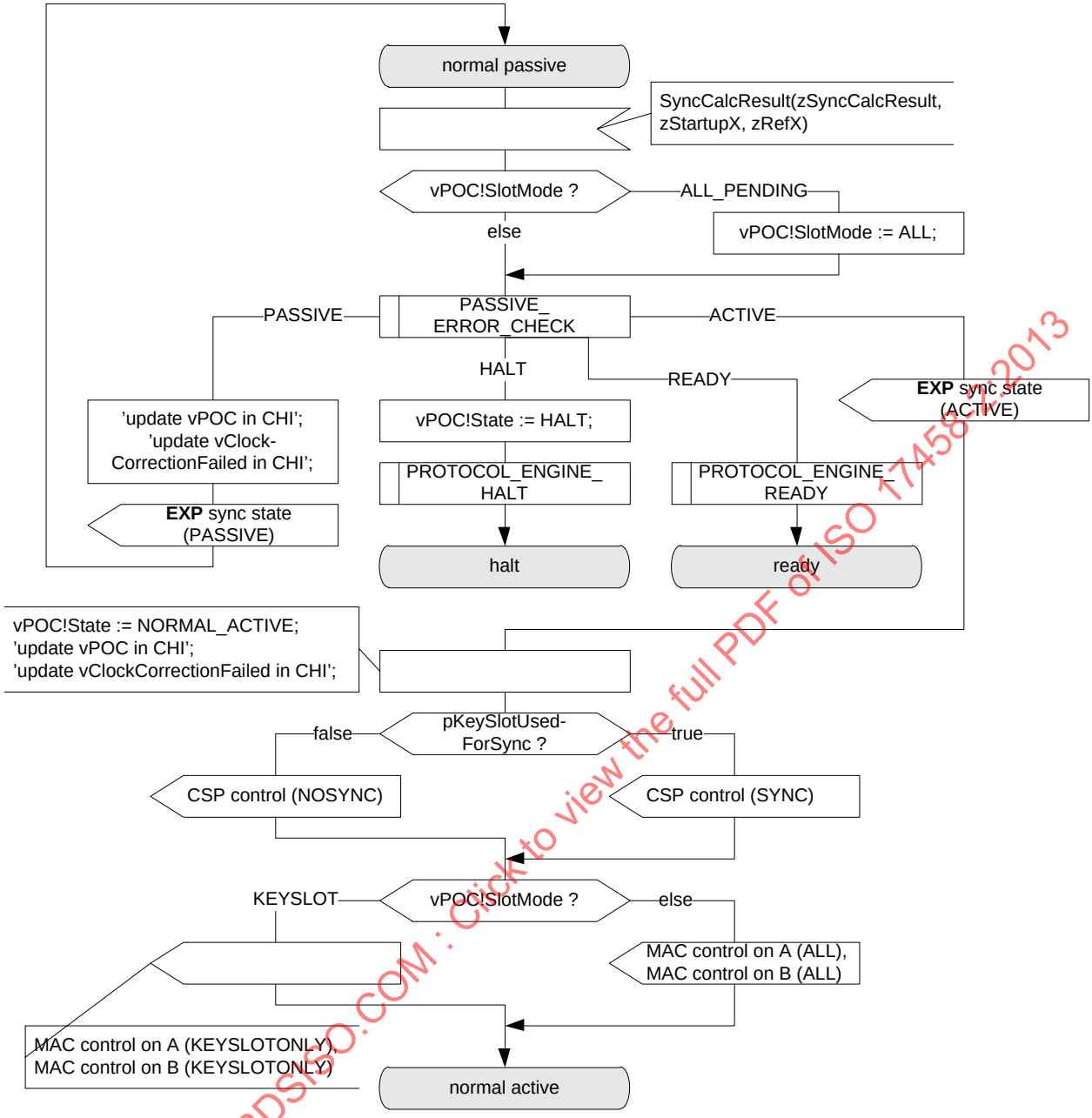



**Figure 34 — Cyclical behaviour in the POC:normal passive state [POC][22)]**



### 6.3.8.2.5 Error checking during normal operation

#### 6.3.8.2.5.1 Error checking overview

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 35) and PASSIVE_ERROR_CHECK (see Figure 36) macros. In both cases, the macro determines a

22) *zStartupX* is *zStartupNodes* in even cycles and *zRxStartupPairs* in odd cycles. *zRefX* is *zRefNode* in even cycles and *zRefPair* in odd cycles. See Figure 157 for details.

new value of *vPOC!ErrorMode* which, in turn, determines the subsequent cycle-end behaviour (see subclauses 6.3.8.2.3 and 6.3.8.2.4).

At the end of each communication cycle CSP communicates the error consequences of the clock synchronisation 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:

— WITHIN_BOUNDS indicates that the calculations resulted in no errors;

— MISSING_TERM indicates that either the rate or offset correction could not be calculated;

— 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 behaviour is detailed in subclauses 6.3.8.2.5.2 and 6.3.8.2.5.3. A number of configuration alternatives and the need to verify cycle timing before resuming communication influence the detailed error checking behaviour. However, the basic concept can be grasped by considering the behaviour in the absence of these considerations. In the absence of these influences, the processing path is determined by the value of *zSyncCalcResult* as follows.

— In all cycles (even or odd), *zSyncCalcResult* = EXCEEDS_BOUNDS causes the POC to transition to the *POC:halt* state.

— In odd cycles, *zSyncCalcResult* = WITHIN_BOUNDS causes the POC to stay in, or transition to, the *POC:normal active* state.

— In odd cycles, if *zSyncCalcResult* = MISSING_TERM.

  — The POC will transition to the *POC:halt* state if the MISSING_TERM value has persisted for at least *gMaxWithoutClockCorrectionFatal* odd cycles.

  — 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 *gMaxWithoutClock-CorrectionFatal* odd cycles.

  — Otherwise the POC stays in the *POC:normal active* state.

#### 6.3.8.2.5.2 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 35.

— If *zSyncCalcResult* is WITHIN_BOUNDS, the *vPOC!ErrorMode* remains ACTIVE.

— If *zSyncCalcResult* is EXCEEDS_BOUNDS.

  — 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.

  — 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.

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

— If zSyncCalcResult is MISSING_TERM and the cycle is odd the behaviour is determined by how many consecutive odd cycles (*vClockCorrectionFailed*) have yielded MISSING_TERM.

  — If *vClockCorrectionFailed* < *gMaxWithoutClockCorrectionPassive*, then the *vPOC!ErrorMode* remains ACTIVE.

  — If *gMaxWithoutClockCorrectionPassive* <= *vClockCorrectionFailed* < *gMaxWithoutClockCorrectionFatal*, then the *vPOC!ErrorMode* is set to PASSIVE.

  — If *vClockCorrectionFailed* >= *gMaxWithoutClockCorrectionFatal* and

    — 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.

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

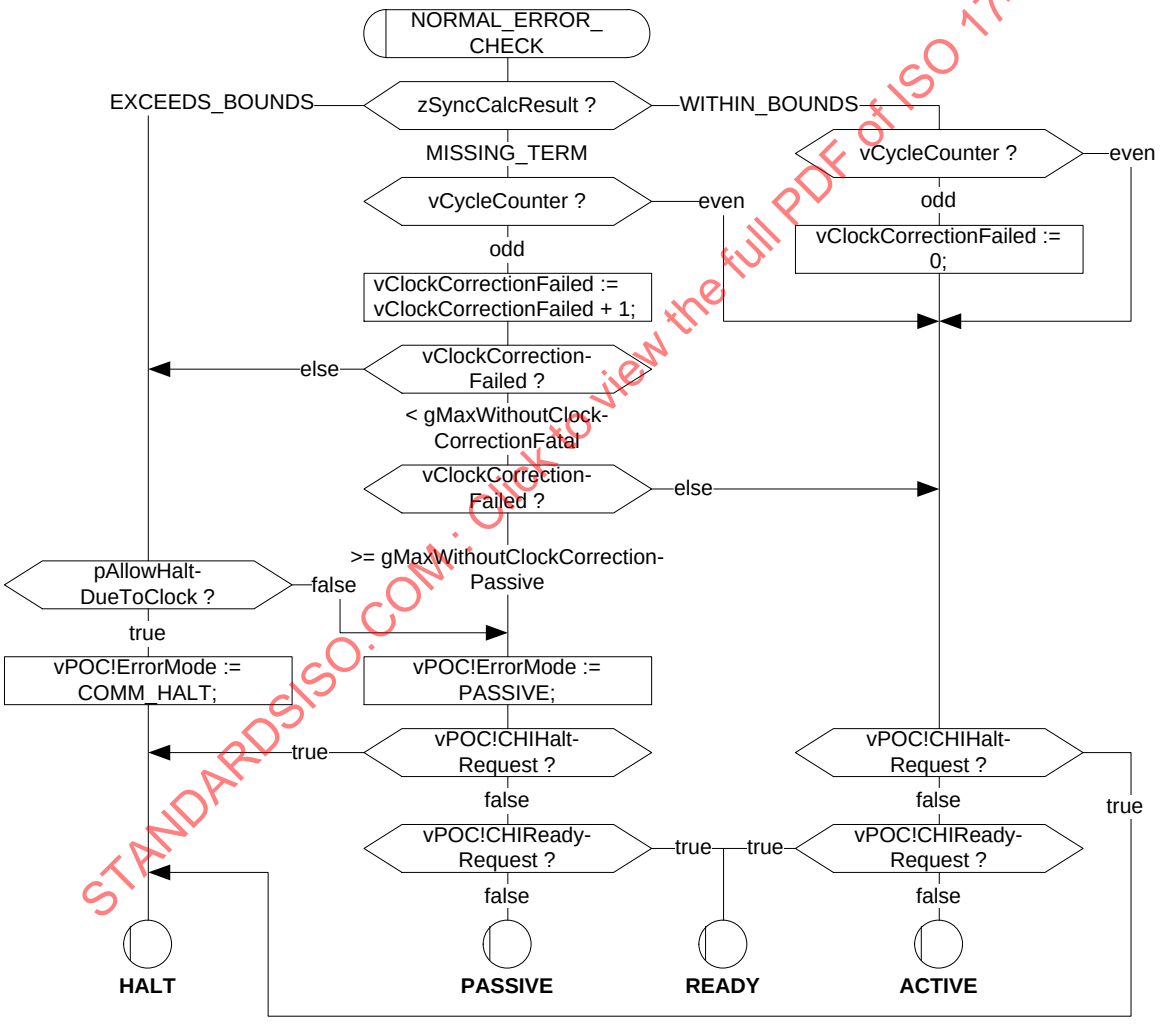



**Figure 35 — Error checking in the POC:normal active state [POC]**

#### 6.3.8.2.5.3 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 36.

- If *zSyncCalcResult* is WITHIN_BOUNDS and it is an even cycle the condition is ignored and *vPOC!ErrorMode* remains PASSIVE.
- If *zSyncCalcResult* is WITHIN_BOUNDS and it is an odd cycle.
  - 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.
  - If the node is configured to allow the resumption of transmissions following the entry to *POC:normal passive* (*pAllowPassiveToActive* is non-zero) the behaviour is determined by how many consecutive odd cycles have yielded WITHIN_BOUNDS.
    - If less than *pAllowPassiveToActive* consecutive odd cycles have yielded WITHIN_BOUNDS, then the *vPOC!ErrorMode* remains PASSIVE.
    - If at least *pAllowPassiveToActive* consecutive odd cycles have yielded WITHIN_BOUNDS, the *vPOC!ErrorMode* depends on the number (*zStartupX*) of startup frame pairs observed in the preceding double cycle.
      - If the node has seen more than one startup frame pair (*zStartupX* > 1) then the *vPOC!ErrorMode* is set to ACTIVE.
      - If the node has seen only one startup frame pair (*zStartupX* = 1) and if the node is a coldstart node (*pKeySlotUsedForStartup* = true) then the *vPOC!ErrorMode* is set to ACTIVE.
      - If neither of the preceding two conditions is met then the *vPOC!ErrorMode* remains PASSIVE.
- If *zSyncCalcResult* is EXCEEDS_BOUNDS:
  - 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.
  - 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.
- If *zSyncCalcResult* is MISSING_TERM and the cycle is even, the condition is ignored and *vPOC!ErrorMode* remains PASSIVE.
- If *zSyncCalcResult* is MISSING_TERM and the cycle is odd, then the behaviour is determined by how many consecutive odd cycles have yielded MISSING_TERM.
  - If at least *gMaxWithoutClockCorrectionFatal* consecutive odd cycles have yielded MISSING_TERM, and
    - 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.
    - 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.

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

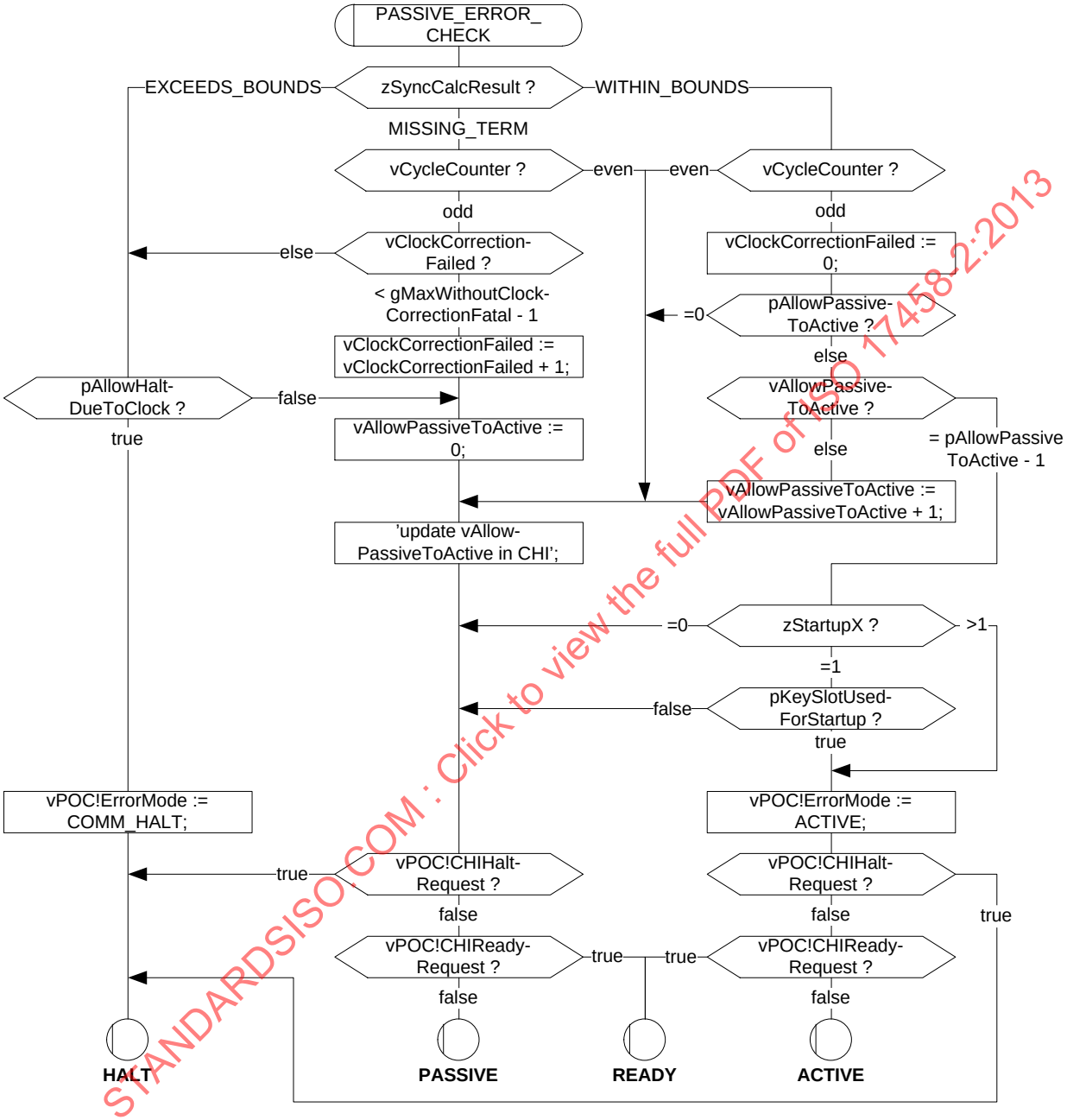





**Figure 36 — Error checking in the POC:normal passive state [POC][23)]**

23) *zStartupX* is *zStartupNodes* in even cycles and *zRxStartupPairs* in odd cycles. *zRefX* is *zRefNode* in even cycles and *zRefPair* in odd cycles. See Figure 157 for details.

# 7 Coding and Decoding

## 7.1 Principles

This subclause describes the coding and decoding behaviour 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 5.8.4 for additional information on the interface between the CC and the BD. The description in this subclause assumes a physical layer that appears to the protocol engine as a binary medium with two distinct levels, called HIGH and LOW[24]. A bit stream generated from these two levels is called a communication element (CE).

A node shall use a non-return to zero (NRZ) signalling 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.

## 7.2 Description

In order to support two channels each node shall implement two sets of independent coding and decoding processes, one for channel A and another for channel B. The subsequent paragraphs of this subclause 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 behaviour is contained in three processes. These processes are the main coding and decoding process (CODEC) and the following two sub-processes:

— Bit strobing process (BITSTRB);

— Wakeup pattern decoding process (WUPDEC).

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

The relationships between the coding / decoding and the other core mechanisms are depicted in Figure 37[26].



24) Detailed bus state definitions may be found in ISO 17458-4.

25) When a dual channel device is configured to operate in a single channel mode, the TxEN and TxD outputs shall be driven high on the unused channel. This requirement applies whenever the "logical" TxEN and TxD outputs are connected to physical pins. It is possible, for example, that an implementation allows more than one function to be combined on a physical pin. In such implementations, this specification places no requirements on the behaviour of those pins when they are not configured to be outputs of the FlexRay protocol.

26) The dark lines represent data flows between mechanisms that are relevant to this subclause. The lighter gray lines are relevant to the protocol, but not to this clause.

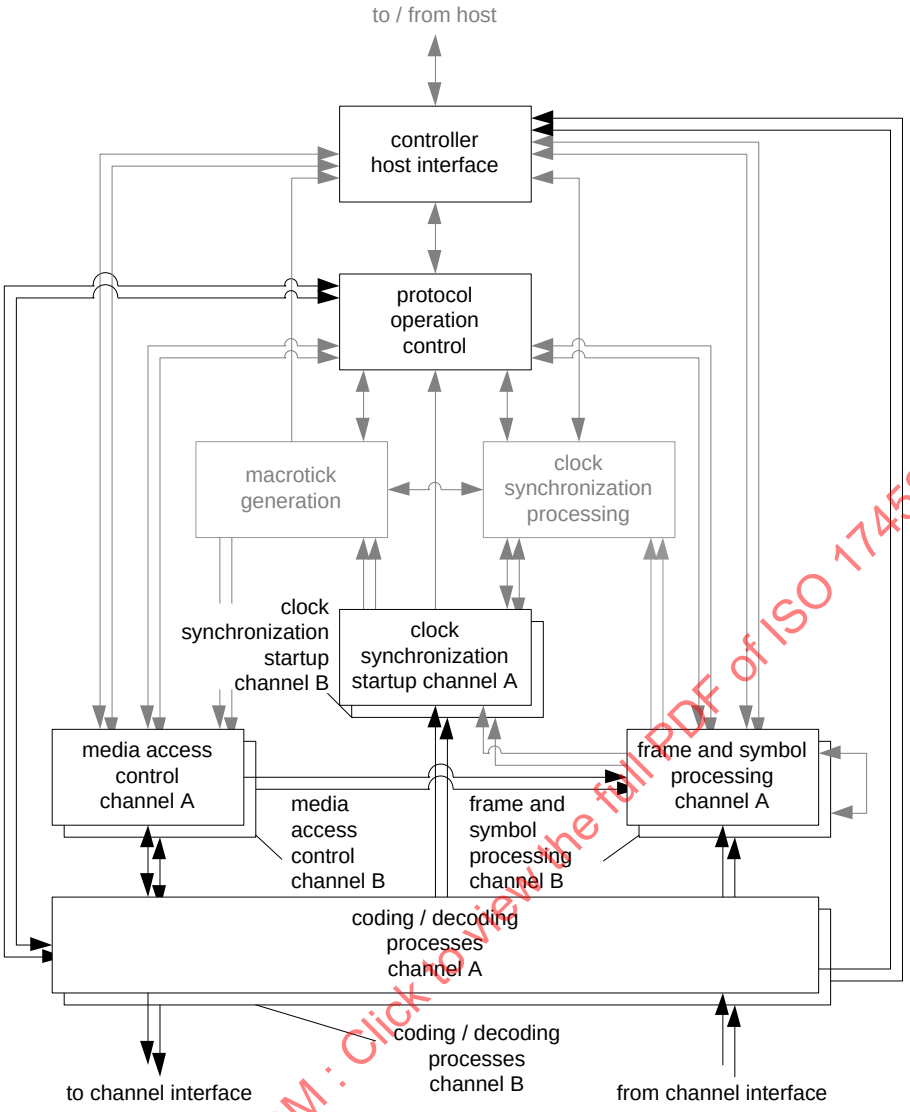



**Figure 37 — Coding / Decoding context**



### 7.2.1 Frame and symbol encoding

#### 7.2.1.1 General

This subclause specifies the behaviour 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.

#### 7.2.1.2 Frame encoding

##### 7.2.1.2.1 Transmission start sequence

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

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

#### 7.2.1.2.2 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.

#### 7.2.1.2.3 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.

#### 7.2.1.2.4 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).

#### 7.2.1.2.5 Dynamic trailing sequence

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

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



27) A TSS is not used before transmission of a WUS or WUDOP because those symbols begin with a sufficiently long active low phase to achieve these goals.
28) See also clause 13.
29) See also 7.2.5.
30) This ensures that there is a period of one *gdBit* during which the TxD output is at the HIGH level prior to the transition of TxEN output to the HIGH level. This is required for the stability of certain types of physical layers.

NOTE The processes defining the behaviour 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.

#### 7.2.1.2.6 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 behaviour, which is described by the CODEC process (see Figure 55) consists of the following steps:

a) Break the frame data down into individual bytes.

b) Prepend a TSS at the start of the bit stream.

c) Add an FSS at the end of the TSS.

d) Create extended byte sequences for each frame data byte by adding a BSS before the bits of the byte.

e) Assemble a continuous bit stream for the frame data by concatenating the extended byte sequences in the same order as the frame data bytes.

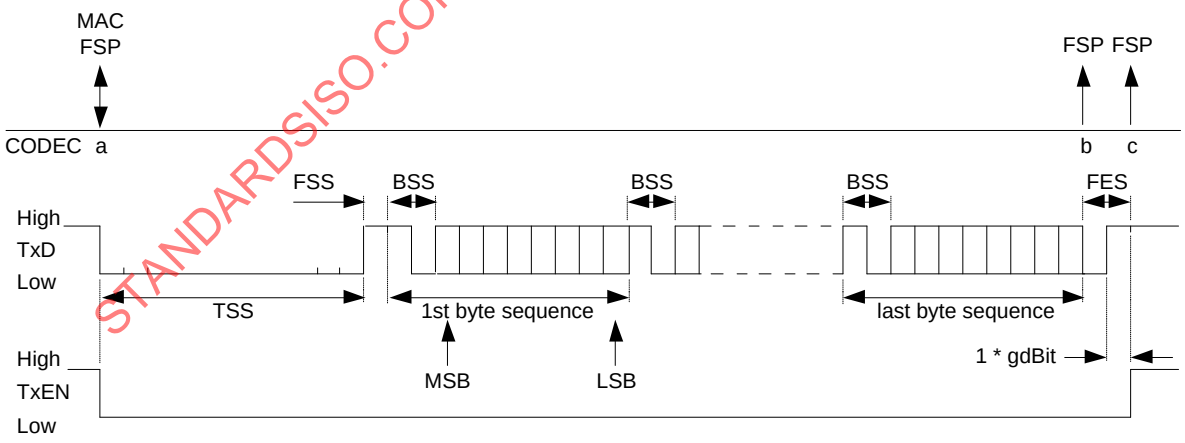
f) Calculate the bytes of the frame CRC, create extended byte sequences for these bytes, and concatenate them to form a bit stream for the frame CRC.

g) Append an FES at the end of the bit stream.

h) Append a DTS after the FES (if the frame is to be transmitted in the dynamic segment).

Steps a) - f) of the list above are performed by the **prepbitstream** function used by the CODEC process (see Figure 55).

Figure 38 shows the bit stream of a frame transmitted in the static segment and related events relevant to the CODEC process:




**Key**

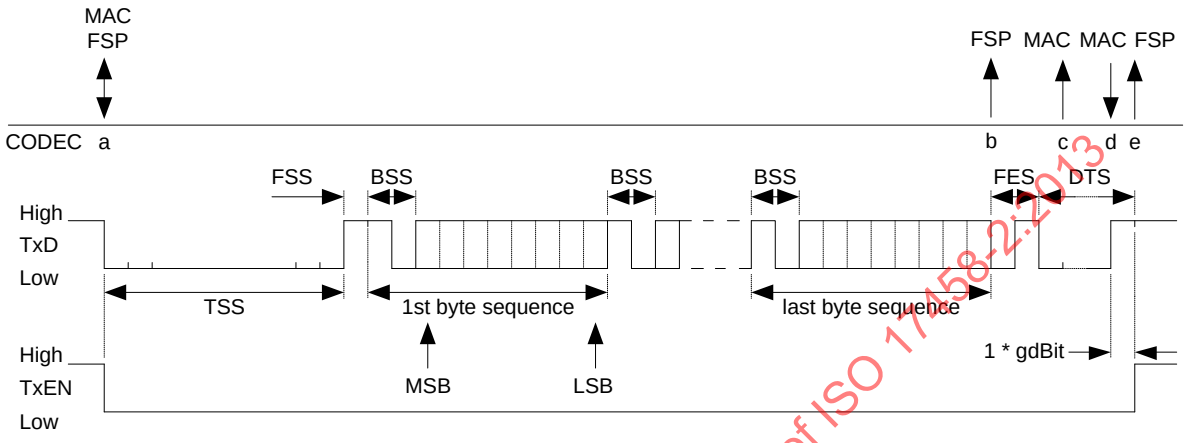
a Input signal transmit frame on A (*vCEType*, *vTF*) received from the MAC process (see Figure 99) and output signal *decoding halted on A* sent to the FSP process (see Figure 119, Figure 120, and Figure 127).

b Output signal frame transmitted on A sent to the FSP process (see Figure 119).

c Output signal decoding started on A sent to the FSP process (see Figure 119).

**Figure 38 — Frame encoding in the static segment**

Figure 39 shows the bit stream of a frame transmitted in the dynamic segment and related events relevant to the CODEC process:




**Key**

a Input signal transmit frame on A (*vCEType*, *vTF*) received from the MAC process (see Figure 104) and output signal *decoding halted on A* sent to the FSP process (see Figure 119, Figure 120, and Figure 127).

b Output signal frame transmitted on A sent to the FSP process (see Figure 128).

c Output signal DTS start on A sent to the MAC process (see Figure 104).

d Input signal stop transmission on A received from the MAC process (see Figure 104).

e Output signal decoding started on A sent to the FSP process (see Figure 128).

**Figure 39 — Frame encoding in the dynamic segment**

### 7.2.1.3 Symbol encoding

#### 7.2.1.3.1 General

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

— Pattern 1 = Collision Avoidance Symbol[31)] (CAS) and Media Access Test Symbol (MTS).

— Pattern 2 = Wakeup Symbol (WUS).

— Pattern 3 = Wakeup During Operation Pattern (WUDOP).

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

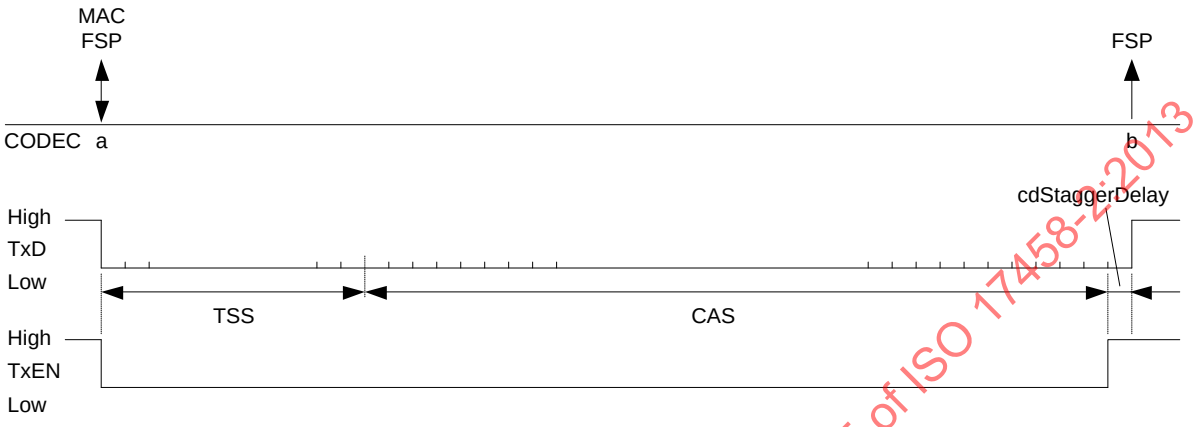
#### 7.2.1.3.2 Collision avoidance symbol and media access test symbol

The node shall transmit these symbols starting with the TSS, followed by a LOW level with a duration of *cdCAS* as shown in Figure 40. The node shall transmit these symbols with the edges of the TxEN signal being synchronous with the TxD at the start of transmission, and with TxD returning to high after a delay period

31) See also clause 11.

(defined by the parameter *cdStaggerDelay*) following the point that the TxEN signal returns to high at the end of transmission. For details refer to Figure 54 and Figure 60[32)].

Figure 40 illustrates the bit stream for a CAS or MTS symbol and related events relevant to the CODEC process:




**Key**

a Input signal transmit symbol on A (*vCEType*) received from the MAC process with *vCEType* = CAS_MTS (from Figure 95 if the node is sending a CAS or from Figure 108 if the node is sending an MTS) and output signal *decoding halted on A* sent to the FSP process (see Figure 119, Figure 120, and Figure 127).

b Output signal decoding started on A sent to the FSP process (see Figure 128).

NOTE See section 3.3 for the used abbreviated terms.

**Figure 40 — CAS and MTS symbol encoding**

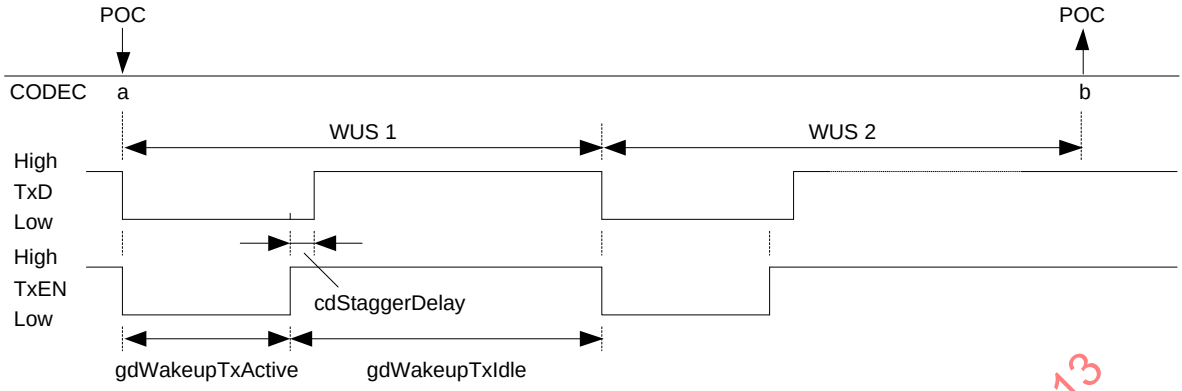


#### 7.2.1.3.3 Wakeup symbol

The node shall support a dedicated wakeup symbol (WUS) composed of *gdWakeupTxActive* bits transmitted at a LOW level followed by *gdWakeupTxIdle* bits of 'idle'. A node generates a wakeup pattern (WUP) by repeating the wakeup symbol *pWakeupPattern* times[33)]. An example of a wakeup pattern formed by a sequence of two wakeup symbols is shown in Figure 41.

32) The delay in TxD at the end of transmission (i.e., a stagger in the deactivation of the TxEN and TxD outputs) is done to avoid the possibility of momentary glitches at the end of symbol transmission that could arise if systematic delays in CC, interface, or BD would cause the TxD signal to effectively transition before the TxEN signal is deactivated. By staggering the deactivation the possibility of such glitches is avoided. A similar situation that could arise at the start of transmission requires no special treatment due to the behavioural characteristics of the BD, where a transmission does not start until both the TxEN and TxD inputs are active (see ISO 17458-4 for further details).

33) *pWakeupPattern* is a configurable parameter that indicates how many times the WUS is repeated to form a WUP. The required value of *pWakeupPattern* is affected by the number of active stars in the communication channel, and by the detailed wakeup forwarding characteristics of these active stars. Details of this configuration are beyond the scope of this document. Refer to ISO 17458-4 and the documentation of the active star components for further details.




**Key**

a Input signal transmit symbol on A (*vCEType*) received from the POC process with *vCEType* = WUP (see Figure 132).

b Output signal WUP transmitted on A sent to the POC process (see Figure 132).

NOTE See section 3.3 for the used abbreviated terms.

**Figure 41 — Wakeup pattern consisting of two wakeup symbols**

The node shall transmit a WUS with the edges of the TxEN signal being synchronous to the TxD at the start of the WUS's low phase, and with TxD returning to high after a delay period (defined by the parameter *cdStaggerDelay*) following the point that the TxEN signal returns to high at the start of the WUS's idle phase (which is also the end of the WUS's low phase)[34].

NOTE There is no TSS transmission associated with a WUS.

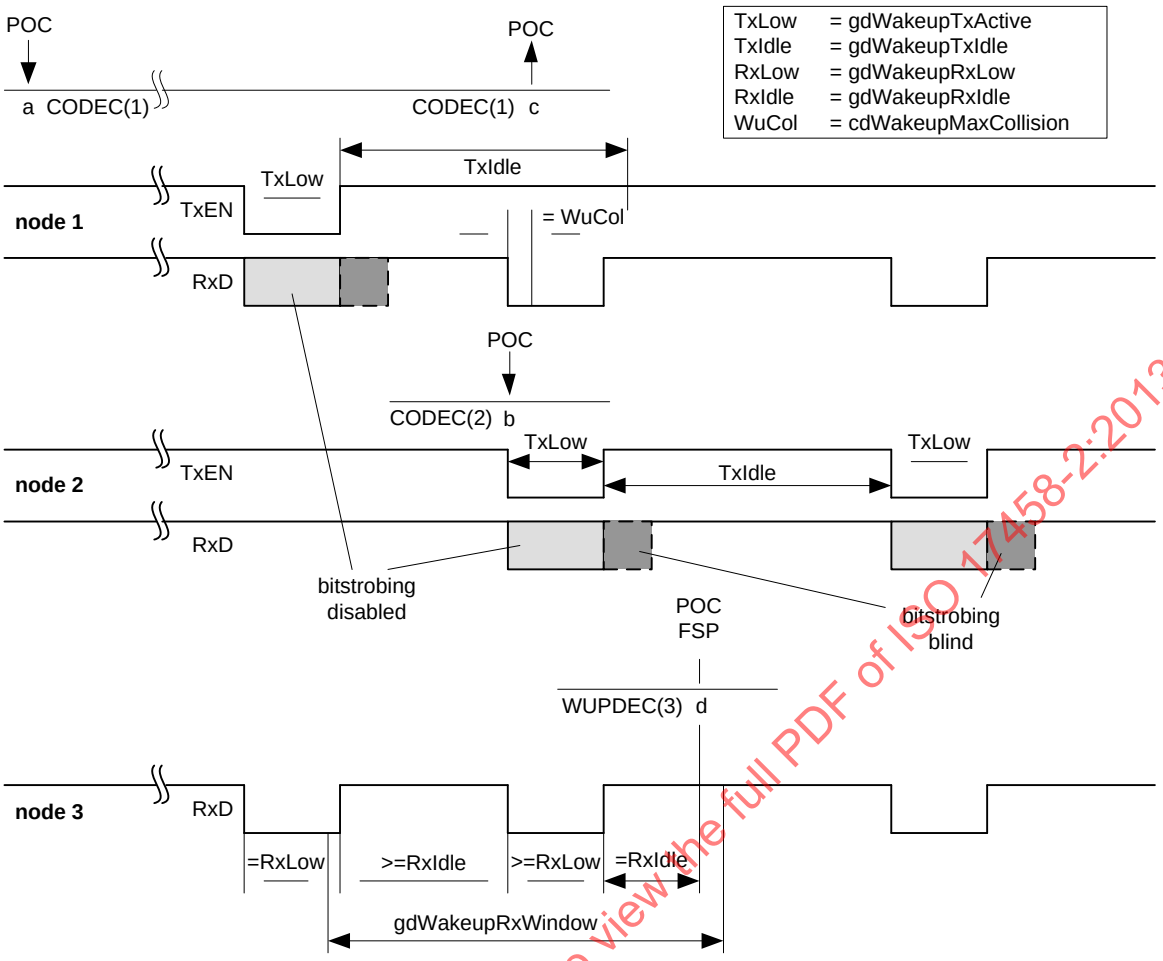
The node shall be capable of detecting activity on the channel during *gdWakeupTxIdle* inside a WUP as shown in Figure 42.

Figure 42 shows an example bit stream that could result from a wakeup symbol collision and shows related events relevant to the CODEC and WUPDEC processes[35].



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

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




**Key**

a Input signal transmit symbol on A (*vCEType*) received from the POC process of node 1 with *vCEType* = WUP (see Figure 132).

b Input signal transmit symbol on A (*vCEType*) received from the POC process of node 2 with *vCEType* = WUP (see Figure 132).

c Output signal wakeup collision on A sent from the CODEC process of node 1 to the POC process of node 1 (see Figure 132).

d Output signal wakeup decoded on A sent from the WUPDEC process of node 3 to the POC process (see 7.2.7.3.2) and to the FSP process (see Figure 117).

**Figure 42 — Wakeup symbol collision and wakeup pattern reception**

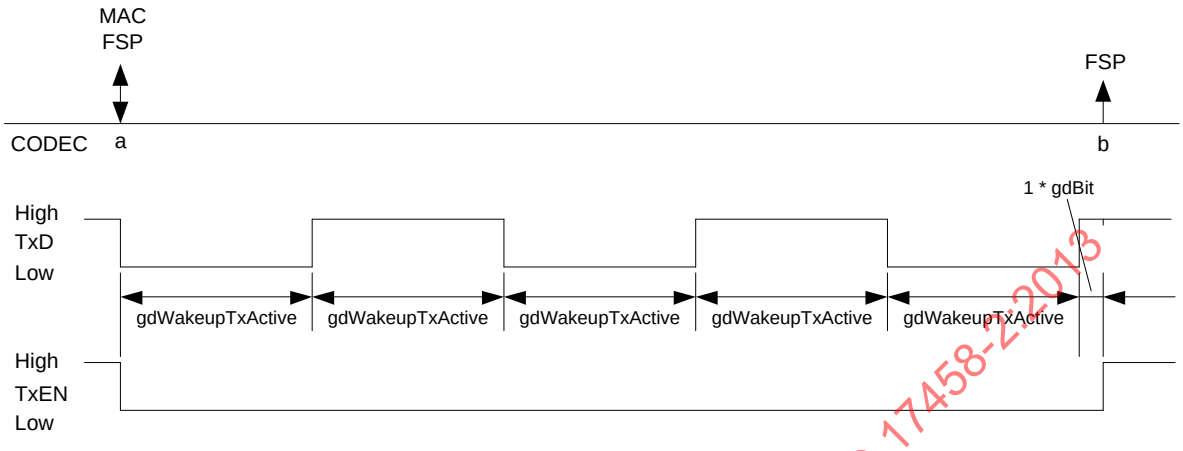


#### 7.2.1.3.4 Wakeup During Operation Pattern (WUDOP)

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

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

Figure 43 shows the bit stream of a WUDOP and related events relevant to the CODEC process:




**Key**

a Input signal *transmit symbol on A* (*vCEType*) received from the MAC process with *vCEType* = WUDOP (see Figure 108) and output signal *decoding halted on A* sent to the FSP process (see Figure 119, Figure 120, and Figure 127).

b Output signal decoding started on A sent to the FSP process (see Figure 128).

**Figure 43 — Wakeup During Operation Pattern**

The node shall transmit a WUDOP with the leading edge of the TxEN signal synchronous to the TxD at the start of the WUDOP's first low phase, and with TxD returning to high one bit time before the TxEN signal returns to high at the end of the WUDOP.

NOTE There is no TSS transmission associated with a WUDOP.

Before and after the actively transmitted WUDOP, a receiver shall see phases of idle on the bus. This is achieved by an appropriate configuration of the symbol window action point (see B.4.11) and the size of the symbol window (see B.4.14).

### 7.2.2 Sampling and majority voting

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

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

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 74.

Figure 44 depicts a sampling and majority voting example. A rising edge on the channel sample clock causes the current value from the RxD bit stream to be sampled and stored within a stabilizing flip-flop. The next rising edge on the sample clock shifts the value from the stabilizing flip-flop into the voting window. The majority of samples within the voting window determines the *zVotedVal* output; the level of *zVotedVal* changes as this majority changes. Single glitches that affect only one or two channel sample clock periods are suppressed. In the absence of glitches, the value of *zVotedVal* has a fixed delay of *cVotingDelay* + *adInternalRxDelay*[38)] sample clock periods relative to the value of the sampled RxD.

All other mechanisms of the decoding process shall consume the signal *bit strobed on A (zVotedVal)* generated by the BITSTRB process (see Figure 74) and shall not directly consider the RxD serial data input signal.

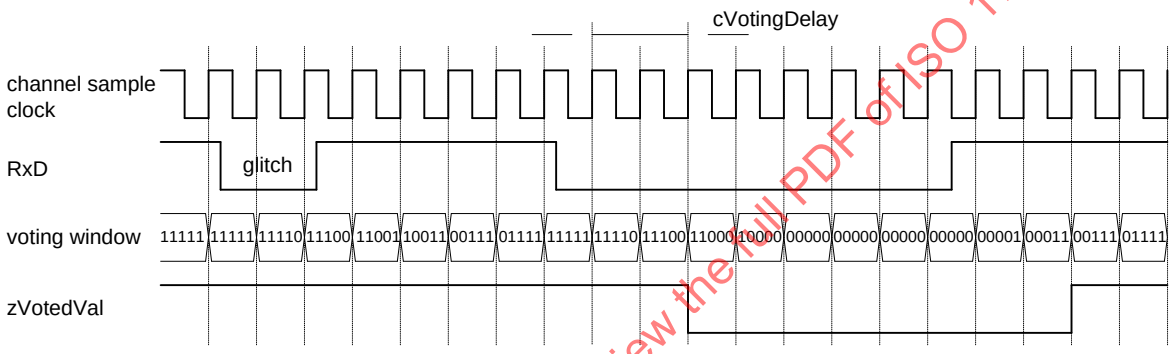



**Figure 44 — Sampling and majority voting of the RxD input (*adInternalRxDelay* = 1)**



### 7.2.3 Bit clock alignment and bit strobing

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

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

A bit synchronisation edge is used to realign the bit timing of the receiver (i.e., bit clock resynchronisation). The node shall enable the bit synchronisation 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. Synchronisation is enabled for the edge between the two bits in the BSS for these bytes.

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

When a bit synchronisation edge is detected (and bit synchronisation 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

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

only perform bit synchronisation on HIGH to LOW transitions of *zVotedVal* (i.e., on the falling edge of the majority voted samples)[39].

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

The bit synchronisation 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 synchronisation. 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 45 depicts the mechanism of the bit synchronisation when a frame is received.

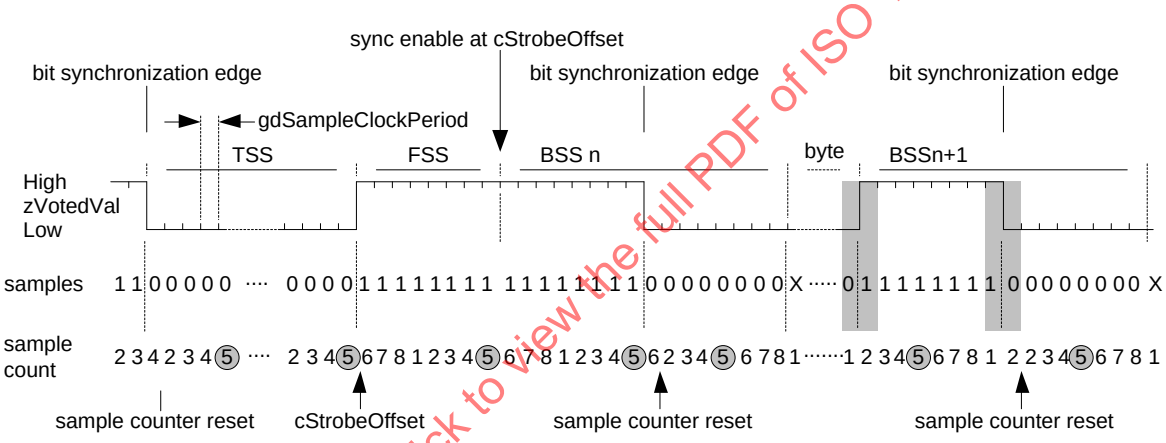



**Figure 45 — Bit synchronisation**

The example from Figure 45 shows the nominal case of an FSS and BSS with *cSamplesPerBit* samples. At the bit synchronisation 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 synchronisation, performed by resetting the sample counter, takes place with the next bit synchronisation edge (see second highlighted area). The effect of the bit synchronisation 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 7.2.8).

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

The SDL representation of the BITSTRB process is depicted in Figure 74.

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

### 7.2.4 Implementation specific delays

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

Such an additional delay might for example be the sum of delays introduced by stabilizing flip-flops after the RXD input or an additional flip-flop after the voting mechanism. The FlexRay data link layer specification tolerates such an additional delay in the range from *cdInternalRxDelayMin* to *cdInternalRxDelayMax* sample clock periods (see B.4.3.2.1). The important conformance test criteria is that the overall delay in this path is in the range defined by the sum of *cVotingDelay* + *cStrobeOffset* + *adInternalRxDelay*.

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

### 7.2.5 Channel idle detection

The node shall use a channel idle detection mechanism to identify the end of the current communication element. Idle detection is done by means of a sample tick timer which is set to a duration of *cChannelIdleDelimiter* times *cSamplesPerBit*. This idle timer is started (or restarted) whenever a bit is strobed as low by the BITSTRB process - expiration of the timer (which implies that no bits have been strobed as low during the duration of the timer[40)] results in the detection of idle (i.e., the channel is considered to be idle).

The channel continues to be considered idle until a bit is strobed as low, which causes the channel to be considered to be active and once again starts the idle timer. Channel idle detection is not active while the node is encoding a communication element - the decoding and encoding mechanisms are mutually exclusive and idle detection is a logical component of the decoding mechanism.

When the CODEC process is instantiated by the POC, the CODEC process immediately instantiates the BITSTRB process and puts it in the GO mode where idle detection is performed. The initial assumption is that the channel is active, so *cChannelIdleDelimiter* times *cSamplesPerBit* sample ticks shall go by without strobing a low bit before the channel is considered to be idle (see 7.4.2).

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

Because of certain effects on the physical layer, a node that has just completed a transmission may experience a period of time where the signal seen at the RxD input does not reflect what is actually occurring on the bus. For example, for a period of time following the completion of a transmission the RxD input may indicate LOW even though no node in the system is actively driving the bus. In order to overcome this issue, when the BITSTRB process is restarted following a transmission it is placed into the BLIND mode.

While operating in this mode, the BITSTRB process basically ignores the actual status of the RxD input and acts as if the RxD input was indicating HIGH at all times. The BITSTRB process remains in the BLIND mode for a configurable number of bit times (determined by the parameter *gdIgnoreAfterTx*) after which it returns to the GO mode where bits are strobed in the normal manner. Although no actual bits are strobed while the BITSTRB process is in the BLIND mode, when BITSTRB leaves the BLIND mode it behaves as if it had strobed *gdIgnoreAfterTx* consecutive HIGH bits.

### 7.2.6 Action point and time reference point

As defined in clause 9, 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.

---

40) This is not necessarily equivalent to strobing *cChannelIdleDelimiter* consecutive high bits because under some circumstances the strobe point can be modified by the bit clock alignment mechanism discussed in 7.2.3.

The clock synchronisation 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 synchronisation algorithm instead infers the time of the transmitter's action point by making a measurement of the arrival time of a received sync frame[41)].

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

Signals transmitted from a node M are received at node N with the propagation delay $dPropagationDelay_{M,N}$. 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 46 depicts the effect of propagation delay and TSS length change. For a more detailed description refer to ISO 17458-4.

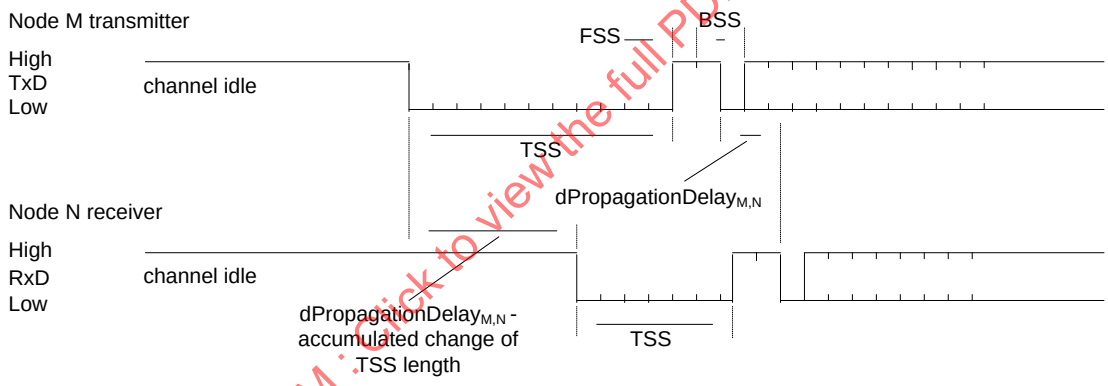



**Figure 46 — TSS length change and propagation**

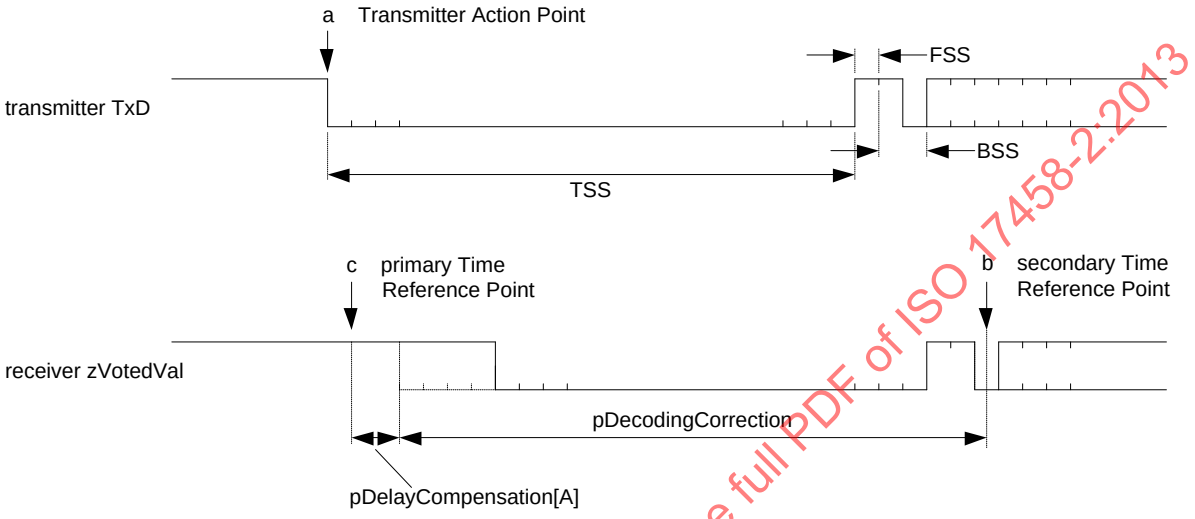
As a result of TSS length change and propagation delay, it is not possible to know the precise relationship between when a receiver begins to see a TSS and when the transmitter started to send the TSS. It is necessary to base the time measurements of received frames on an element of the frame that is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

41) This is possible because transmission of the sync frame begins at the static slot action point of the transmitting node.
42) The use of the "+ 2" constant places requirements on the EPL parameters such as *dFrameTSSLengthChange* and *dFrameTSSEMIInfluence*. See Table B.5 for additional details.

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 47 depicts definitions for the time reference point calculations and shows the following significant events.




**Key**

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 12.4.3).

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. This does not represent an actual event, but rather only indicates the point in time that the timestamp represents.

**Figure 47 — Time reference point definitions**

The difference between *zPrimaryTRP* and *zSecondaryTRP* is the summation of node parameters *pDecodingCorrection* and *pDelayCompensation*. The calculation of *pDecodingCorrection* is given in B.4.26.

The Primary TRP timestamp is passed to the FSP process (and subsequently to the clock synchronisation process) via the PrimaryTRP element of the *vRF* structure (see Figure 72, Figure 118, Figure 120, and Figure 165). The clock synchronisation 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 clause 12.

### 7.2.7 Frame and symbol decoding

#### 7.2.7.1 Overview

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

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

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

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

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

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

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 shall be synchronized, i.e., there shall be an integer multiplicative relationship between the two periods and the two clocks shall have a fixed phase relationship.

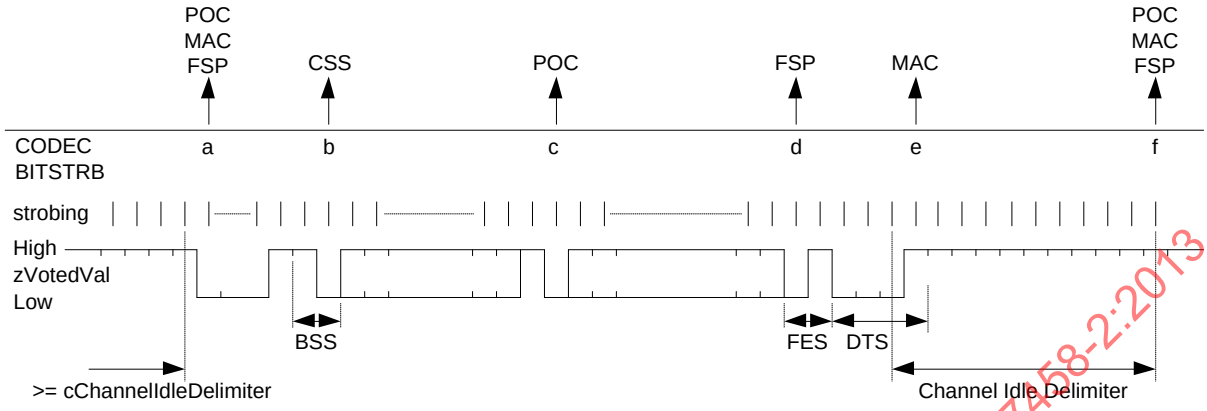
#### 7.2.7.2 Frame decoding

A frame starts at CE start with the first strobed LOW bit after channel idle. The channel idle delimiter refers to the time required by the idle detection mechanism to determine that the channel is idle (see 7.2.5). In order for idle to occur all bits strobed during the channel idle delimiter shall be strobed as high.



43) The successful decoding does not necessarily imply successful reception in terms of being able to present the payload of the decoded stream to the host.

Figure 48 depicts the received frame bit stream of a frame and events in relation to the CODEC and the BITSTRB processes.




**Key**

a Output signal *idle end on A* to the POC process shown in Figure 31, Figure 131, and Figure 142, and to the MAC process shown in Figure 102; output signal *CE start on A* to the MAC process shown in Figure 103, and to the FSP process shown in Figure 119.

b Output signal *potential frame start on A* to the CSS process shown in Figure 162.

c Output signal *header received on A* to the POC process shown in Figure 131, Figure 133, Figure 142, Figure 144, and Figure 146.

d Output signal *frame decoded on A (vRF)* to the FSP process shown in Figure 120.

e Output signal potential idle start on A to the MAC process shown in Figure 103; Output signal DTS received on A to the MAC process shown in Figure 103.

f Output signal CHIRP on A to the MAC process shown in Figure 103 and Figure 102, to the FSP process shown in Figure 127, and to the POC process shown in Figure 31, Figure 131, and Figure 142.

**Figure 48 — Received frame bit stream**

The BITSTRB process shall output a *potential idle start on A* signal every time a bit strobed as high was preceded by a bit strobed as low. To keep the figure simple these signals are not shown in Figure 48 with the exception of the potential idle start at the end of the DTS. Although all of these *potential idle start* signals are processed by the MAC process, only the one associated with the end of the DTS is relevant to the operation of dynamic segment media access.

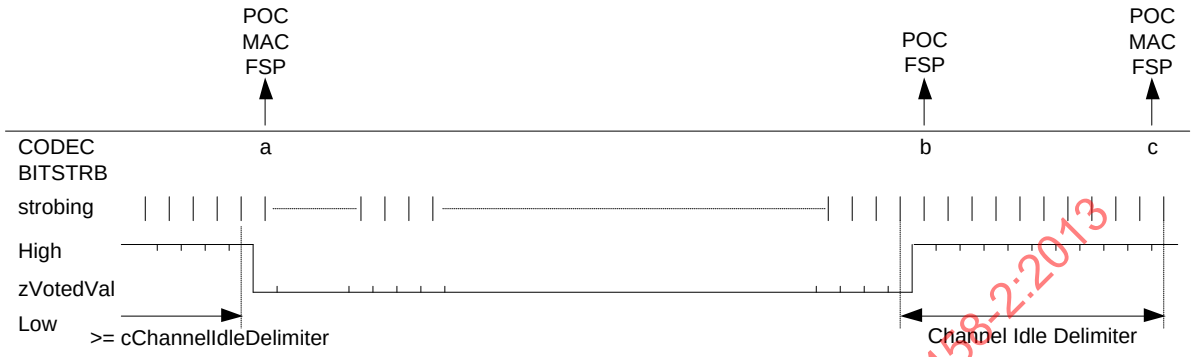
### 7.2.7.3 Symbol decoding

#### 7.2.7.3.1 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 49 shows the received bit stream of a CAS / MTS and events in relation to the CODEC and the BITSTRB processes:




**Key**

a Output signal idle end on A to the POC process shown in Figure 31, Figure 131, and Figure 142, and to the MAC process shown in Figure 102; output signal *CE start on A* to the MAC process shown in Figure 103, and to the FSP process shown in Figure 119.

b Output signal CAS_MTS decoded on A to the POC process shown in Figure 131, Figure 142, Figure 144, and Figure 146, to the MAC process shown in Figure 103, and to the FSP process shown in Figure 121.

c Output signal CHIRP on A to the POC process shown in Figure 31, Figure 131, and Figure 142, to the MAC process shown in Figure 103 and Figure 102, and to the FSP process shown in Figure 127.

**Figure 49 — Received CAS / MTS bit stream**



#### 7.2.7.3.2 Wakeup symbol decoding

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

— a HIGH level with a duration of at least *gdWakeupRxIdle* is detected;

— this is followed by a duration of at least *gdWakeupRxLow* at the LOW level. After this duration a window timer is started, even if the LOW level is still present. The timer will expire after *gdWakeupRxWindow*;

— this is followed by a duration of at least *gdWakeupRxIdle* at the HIGH level;

— this is followed by a duration of at least *gdWakeupRxLow* at the LOW level;

— this is followed by a duration of at least *gdWakeupRxIdle* at the HIGH level;

— the last three phases of HIGH, LOW and HIGH are received before the window timer that was started during the first LOW phase has expired.

The bit stream received by node 3 in Figure 42 shows the reception of a WUP and the related event relevant to the WUPDEC process of node 3. The output signal *wakeup decoded on A* is sent to the POC process shown in Figure 131 and Figure 133, and to the FSP process shown in Figure 117.

#### 7.2.7.4 Decoding error

Exiting one of the decoding macros CAS_MTS_DECODING, FSS_BSS_DECODING, HEADER_DECODING, PAYLOAD_DECODING, or TRAILER_DECODING (shown in Figure 62) 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.

It shall not lead to a decoding error if one or three HIGH bits (instead of exactly two HIGH bits) are strobed after the TSS.

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

— (2 * *cSamplesPerBit* - 1), or

— (2 * *cSamplesPerBit*), or

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

In addition to the effects of quantization, the presence of asymmetry could also result in additional reduction or lengthening of the actual duration of the HIGH period of the FSS-BSS sequence. Any amount of asymmetry that is accommodated by the decoding algorithm would still result in the strobing of between one and three bits at HIGH. Refer to ISO 17458-4 for additional details.

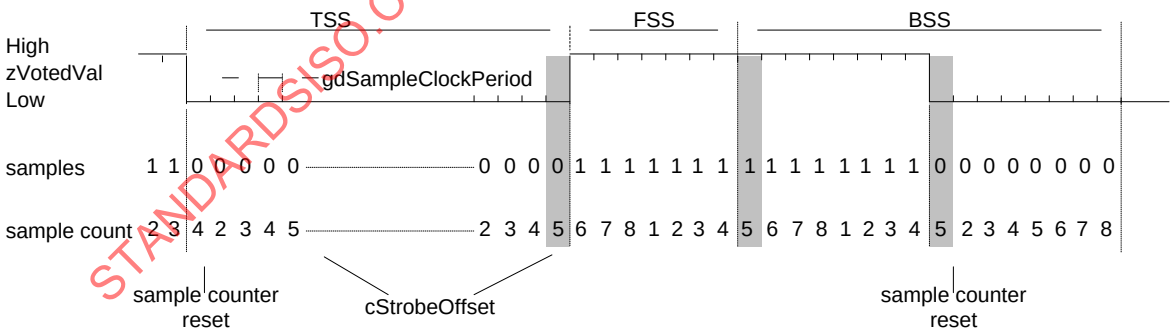



**Figure 50 — Start of frame with FSS BSS decoding**

#### 7.2.8 Signal integrity

In general, there are several conditions (e.g. clock oscillator differences, electrical characteristics of the transmission media or the transceivers, EMI etc.) that can cause variations of signal timing or introduce anomalies / glitches into the communication bit stream. The decoding function attempts to enable tolerance of

the physical layer against presence of one glitch in a bit cell when the length of the glitch is less than or equal to one channel sample clock period[44)].

Asymmetric delays cause bit edges to occur earlier or later than would otherwise be expected from a receiver's perspective. This could contribute to individual receivers incorrectly determining a received bit value. To avoid this effect these delays shall 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 ISO 17458-4.

## 7.3 Coding and decoding process

### 7.3.1 Operating modes

The POC shall set the operating mode of the CODEC for each communication channel.

Definition **(9)** shows the formal definition of the CODEC operating modes.

**Definition: *T_CodecMode*** **(9)**

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

— In the STANDBY mode, the execution of the CODEC and all of its subprocesses are effectively halted.

— In the READY mode the bit strobe process BITSTRB and the wakeup detection process WUPDEC are executed but the CODEC process waits in its *CODEC:ready* state.

— In the NORMAL mode the CODEC process, the bit strobe process BITSTRB and the wakeup detection process WUPDEC are executed.

### 7.3.2 Coding and decoding process behaviour

This subclause contains the formalized specification of the CODEC control process. Figure 51, Figure 52 and Figure 53 depict the specification of the CODEC control process and the termination of the CODEC process. When the CODEC process receives the POC signal *terminate CODEC_A*, the CODEC sends termination signals to its subprocesses before terminating itself.



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

Figure 51 depicts the specification of the CODEC process.

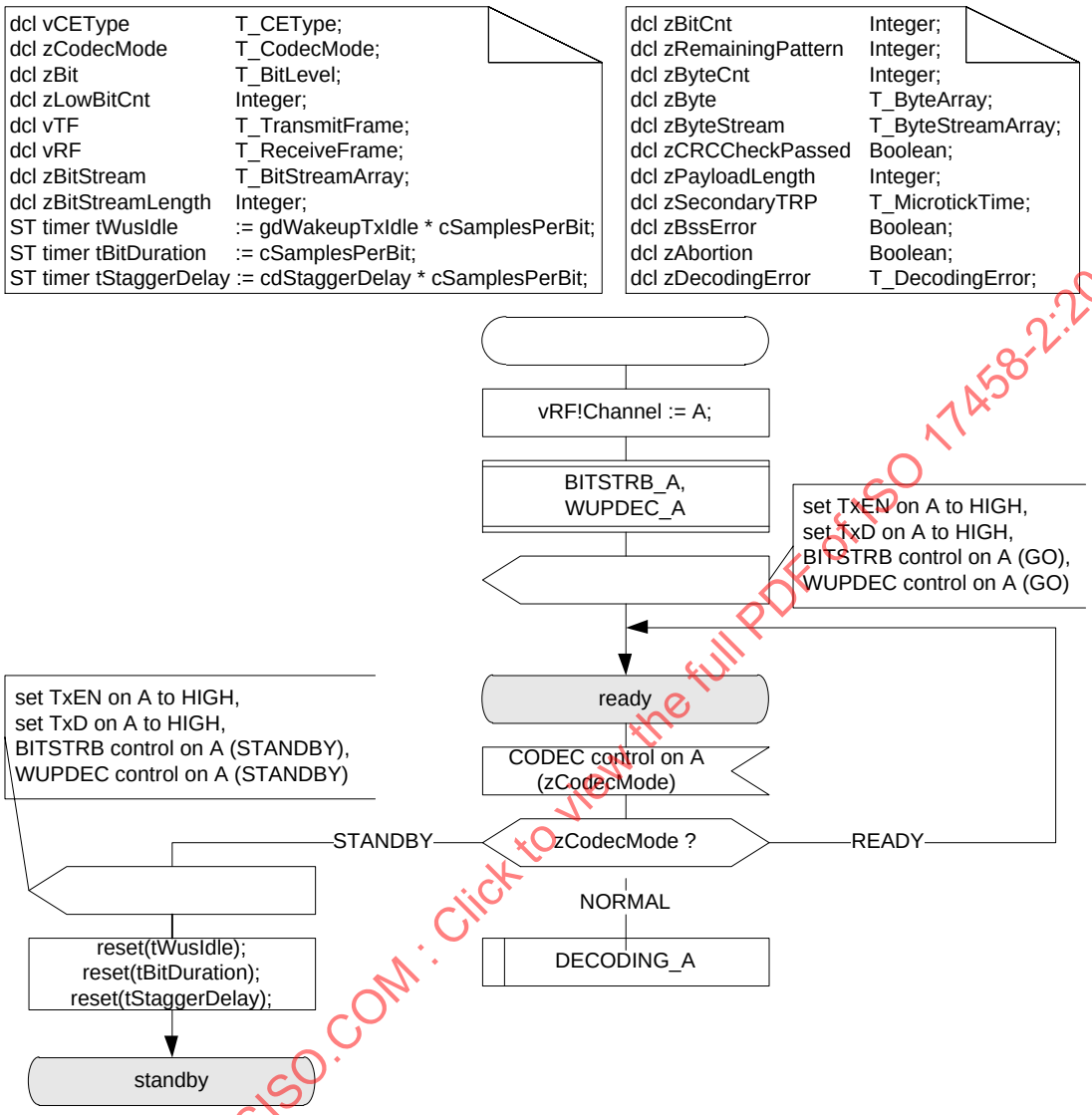



**Figure 51 — CODEC process [CODEC_A]**

Figure 52 depicts the specification of the CODEC control process.

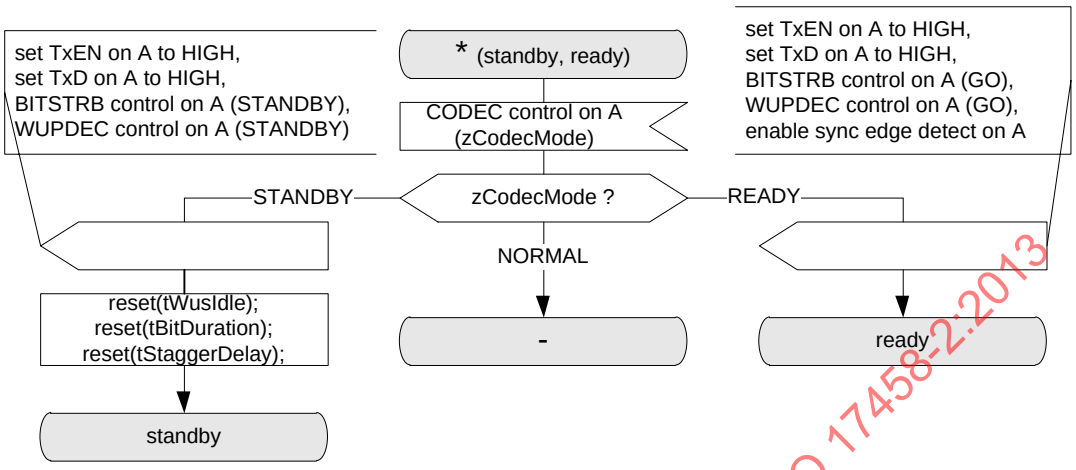



**Figure 52 — Mode control of the CODEC process [CODEC_A]**

Figure 53 depicts the specification of the CODEC termination process.

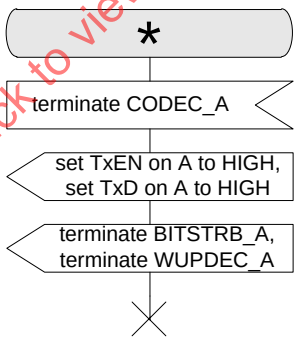



**Figure 53 — Termination of the CODEC process [CODEC_A]**



### 7.3.3 Encoding behaviour

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[45].

**Definition: *T_TransmitFrame*** **(10)**

```
newtype T_TransmitFrame
struct
```

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

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

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*.

**Definition: *T_BitLevel*** **(11)**

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

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

Figure 54 shows a high level view of the mechanisms used for encoding. After the reception of a *transmit frame on A* or *transmit symbol on A* signal the CODEC process follows one of four distinct paths depending on the type of communication element that is to be encoded. Each of these paths is described in a separate macro. Before any of the macros are executed, however, the CODEC process sets the TxEN output to LOW and puts the BITSTRB and WUPDEC processes into STANDBY mode.

If the CE to be encoded is a frame, the FRAME_ENCODING macro is executed (see Figure 55). This macro calls the prepbitstream function, which takes the inbound frame *VTF* from the MAC process, prepares the bit stream *zBitStream* of type *T_BitStreamArray* for transmission, and calculates the bit stream length *zBitStreamLength*. The prepbitstream function shall break the frame data down into individual bytes, prepend a TSS, add an FSS at the end of the TSS, create a series of extended byte sequences by adding a BSS at the beginning of each byte of frame data, calculate the frame CRC bytes and create the extended byte sequences for the CRC, and assemble a continuous bit stream out of the extended byte sequences.

Once the bit stream is prepared, the macro steps bit by bit through the stream and presents it to the bus. Following this, the FRAME_ENCODING macro calls the TRANSMIT_FES macro and, if the frame is sent in the dynamic segment, the TRANSMIT_DTS macro. Once all of these are complete, the macro sends a control signal to the BITSTRB process that restarts bit strobing in the BLIND mode.

If the CE to be encoded is a CAS/ MTS, the CAS_MTS_ENCODING macro is executed (see Figure 60). This macro simply sets the TxD output to LOW and leaves it at this state for an appropriate period of time (the duration of TSS plus the duration of the CAS). Once the necessary duration has passed (as indicated by the WAIT function described in Figure 58) the TxEN output is deactivated and the BITSTRB process is restarted in the BLIND mode. Following a delay of *cdStaggerDelay* bits (see 7.2.1.3.2), the TxD output is also deactivated.

If the CE to be encoded is a Wakeup Pattern, the WUP_ENCODING macro is executed (see Figure 59). This macro loops through *pWakeupPattern* iterations, each one creating a WUS as described in 7.2.1.3.3. As the low phase of each WUS is completed, the WUP_ENCODING macro provides the necessary staggering of the TxD and TxEN outputs, and commands the BITSTRB process to the BLIND mode to begin the process of allowing the detection of wakeup symbol collisions. If a collision is detected (by detecting *cdWakeupMaxCollision* consecutive LOW bits) the WUP_ENCODING macro is aborted, signalling a wakeup collision. If no collision is detected, the macro will loop until all WUS's have been generated and then exit.

If the CE to be encoded is a WUDOP, the WUDOP_ENCODING macro is executed (see Figure 61). This macro transmits a sequence of LOW-HIGH-LOW-HIGH-LOW phases followed by a brief HIGH phase. Following the last HIGH phase the TxEN output is deactivated and the BITSTRB process is restarted in the BLIND mode.

**Definition: *T_BitStreamArray*** **(12)**

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

Figure 54 depicts the encoding mechanism [CODEC_A].

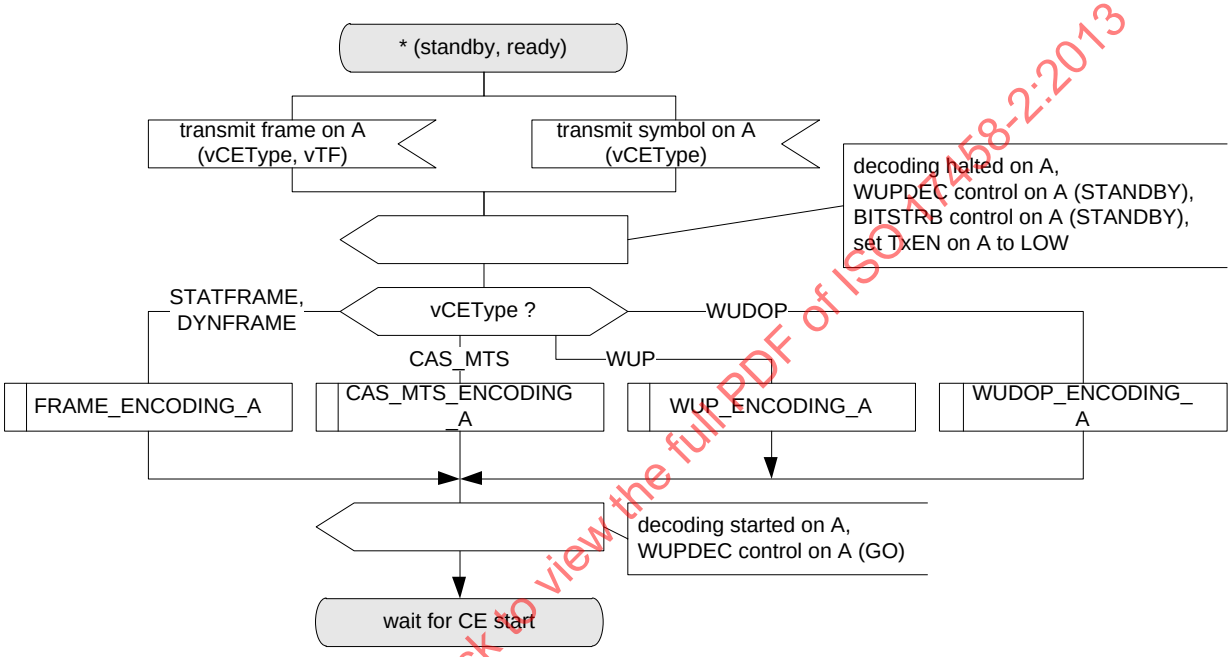



**Figure 54 — Encoding mechanism [CODEC_A]**

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

**Definition: *T_CEType*** **(13)**

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

### 7.3.4 Encoding macros

Figure 55 depicts the frame encoding macro FRAME_ENCODING_A [CODEC_A].

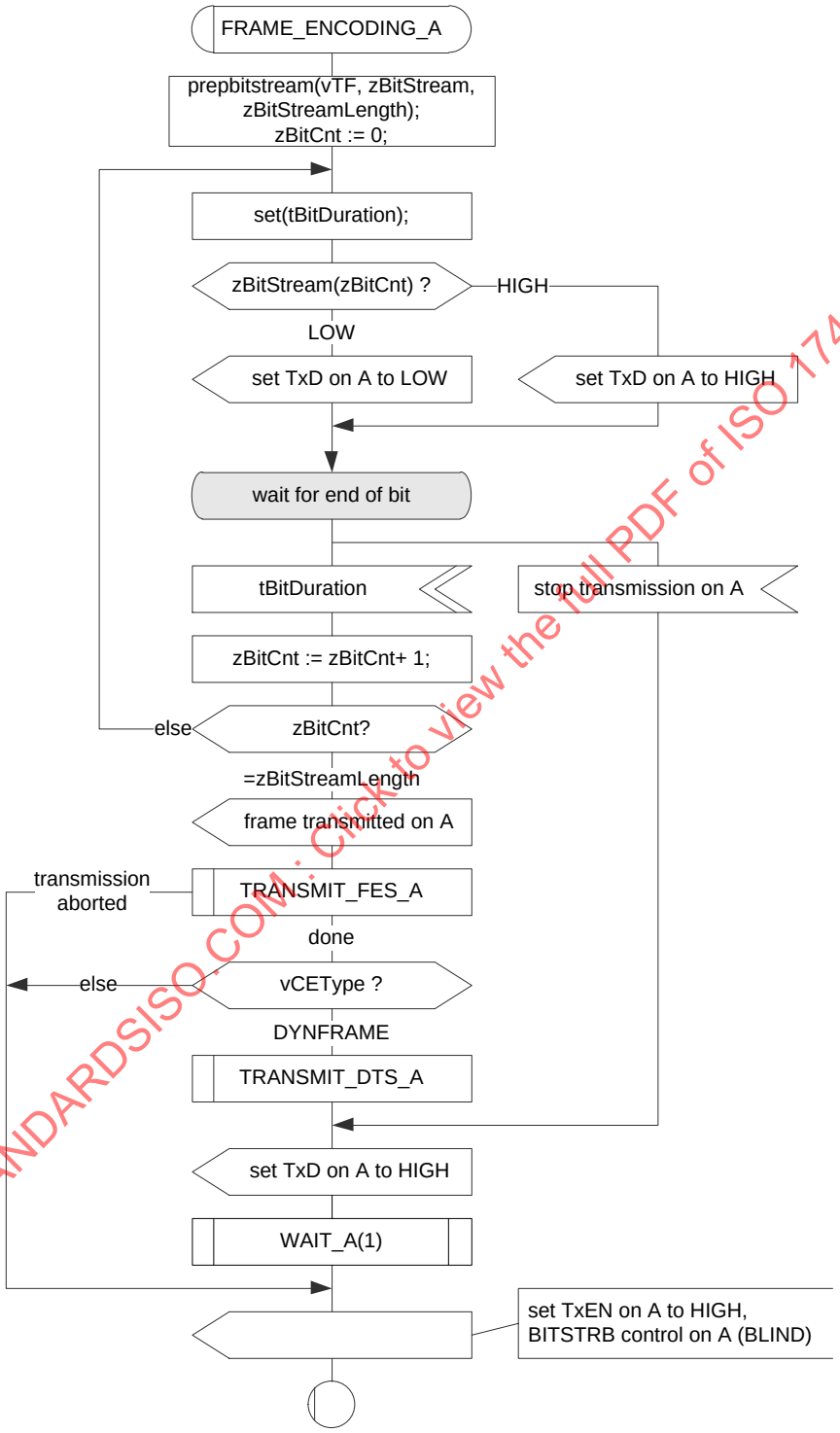



**Figure 55 — Encoding macro FRAME_ENCODING_A [CODEC_A]**

Figure 56 depicts the FES encoding macro TRANSMIT_FES_A [CODEC_A].

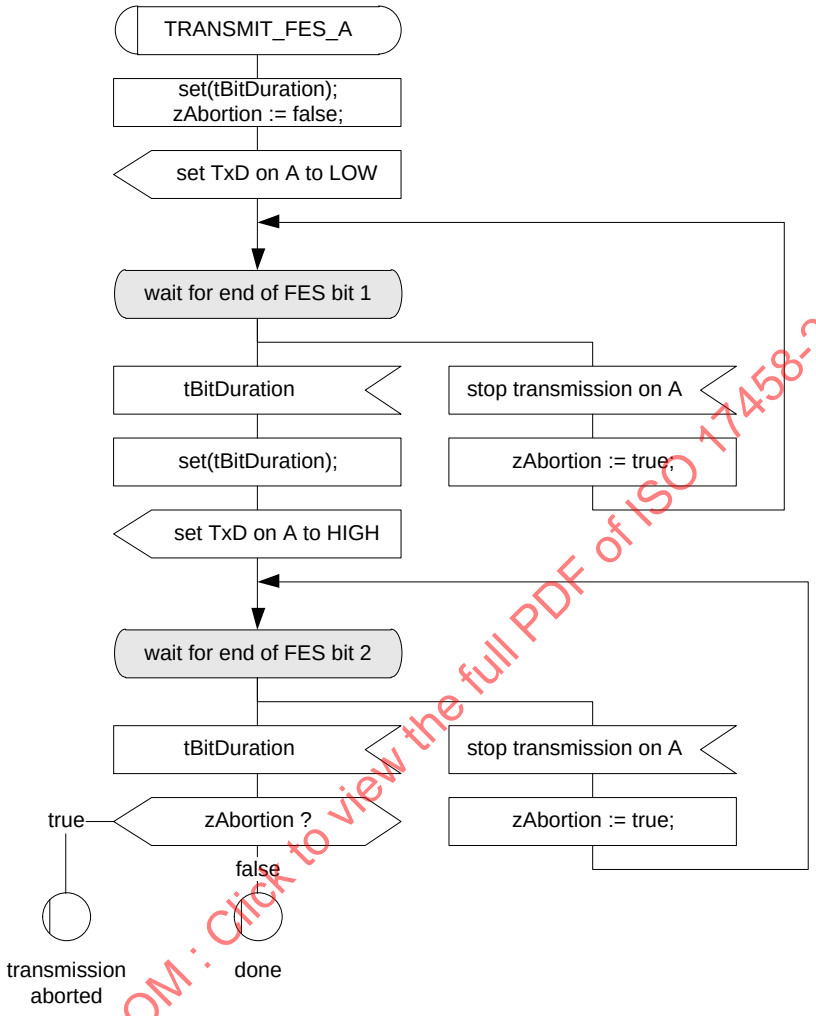



**Figure 56 — Encoding macro TRANSMIT_FES_A [CODEC_A]**

Figure 57 depicts the DTS encoding macro TRANSMIT_DTS_A [CODEC_A].

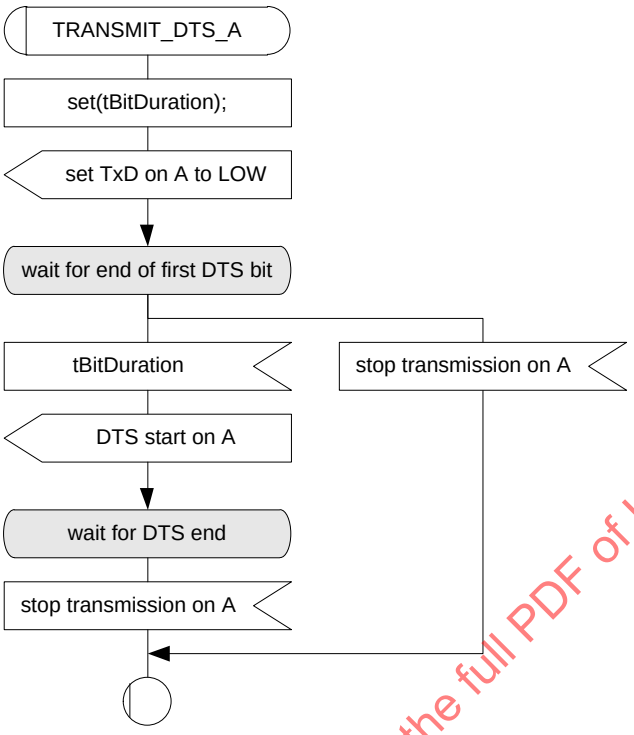



**Figure 57 — Encoding macro TRANSMIT_DTS_A [CODEC_A]**

Figure 58 depicts the procedure WAIT_A [CODEC_A].

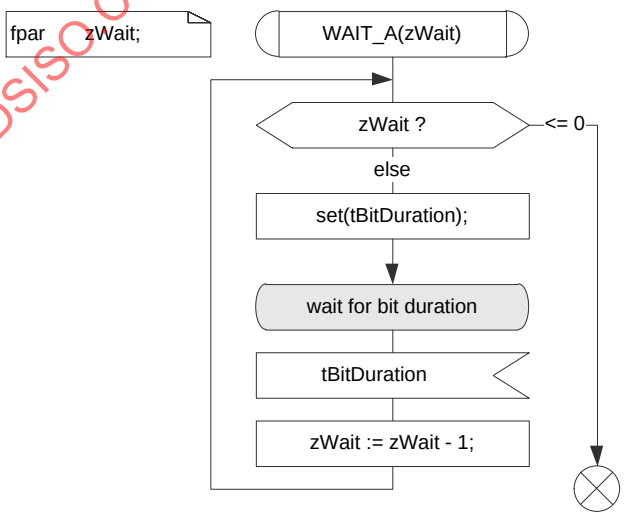



**Figure 58 — Procedure WAIT_A [CODEC_A]**

Figure 59 depicts the WUP encoding macro WUP_ENCODING_A [CODEC_A].

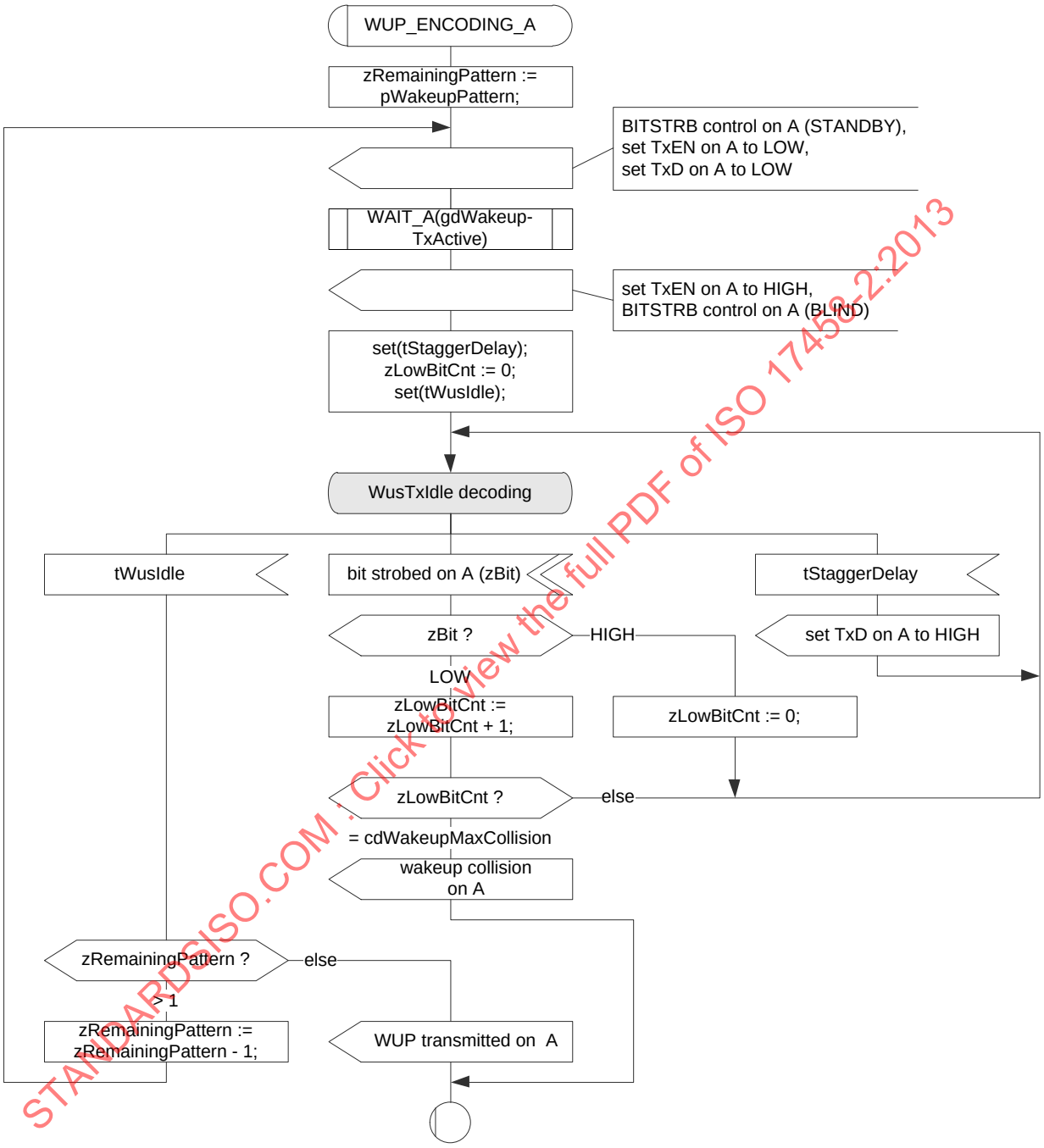





**Figure 59 — Encoding macro WUP_ENCODING_A [CODEC_A]**

Figure 60 depicts the CAS / MTS encoding macro CAS_MTS_ENCODING_A [CODEC_A].

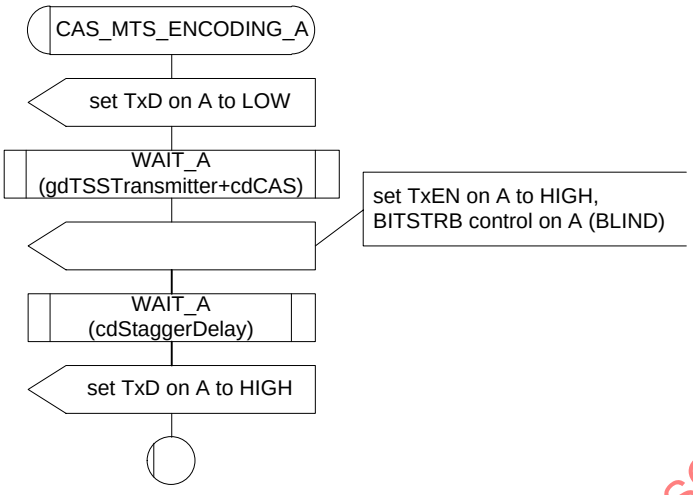



**Figure 60 — Encoding macro CAS_MTS_ENCODING_A [CODEC_A]**

Figure 61 depicts the WUDOP encoding macro WUDOP_ENCODING_A [CODEC_A].


**Figure 61 — Encoding macro WUDOP_ENCODING_A [CODEC_A]**



### 7.3.5 Decoding behaviour

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

Figure 62 depicts the macro DECODING_A [CODEC_A].

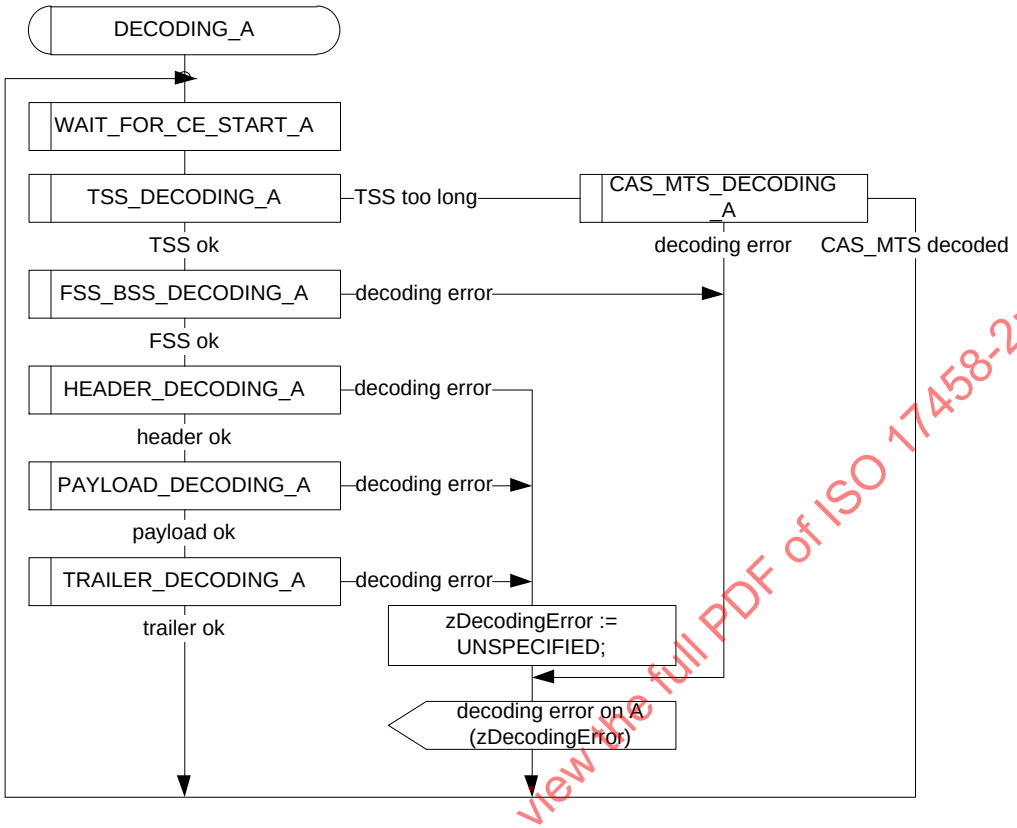



**Figure 62 — Decoding macro DECODING_A [CODEC_A]**



### 7.3.6 Decoding macros

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

**Definition: *T_ByteArray*** **(14)**

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

**Definition: *T_ByteStreamArray*** **(15)**

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

**Definition: *T_CRCCheckPassed*** **(16)**

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

**Definition: *T_MicrotickTime*** **(17)**

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

**Definition: *T_ReceiveFrame*** **(18)**

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

**Definition: *T_DecodingError*** **(19)**

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

Figure 63 depicts the macro WAIT_FOR_CE_START_A [CODEC_A].

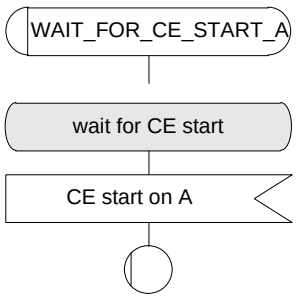



**Figure 63 — Decoding macro WAIT_FOR_CE_START_A [CODEC_A]**

Figure 64 depicts the macro TSS_DECODING_A [CODEC_A].

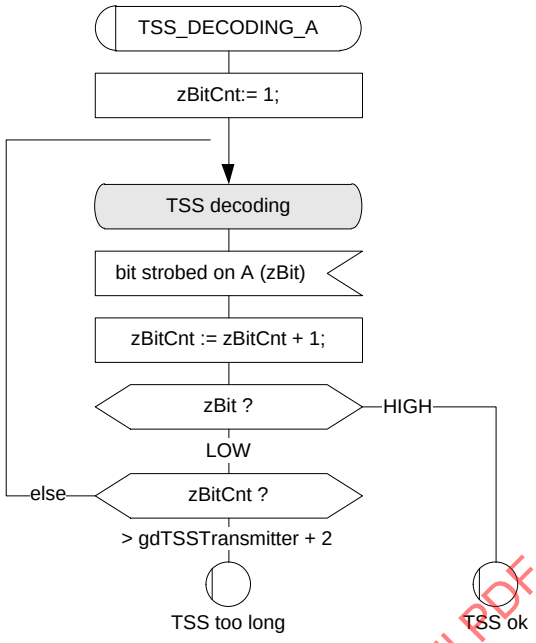



**Figure 64 — Decoding macro TSS_DECODING_A [CODEC_A]**

Figure 65 depicts the macro CAS_MTS_DECODING_A [CODEC_A].

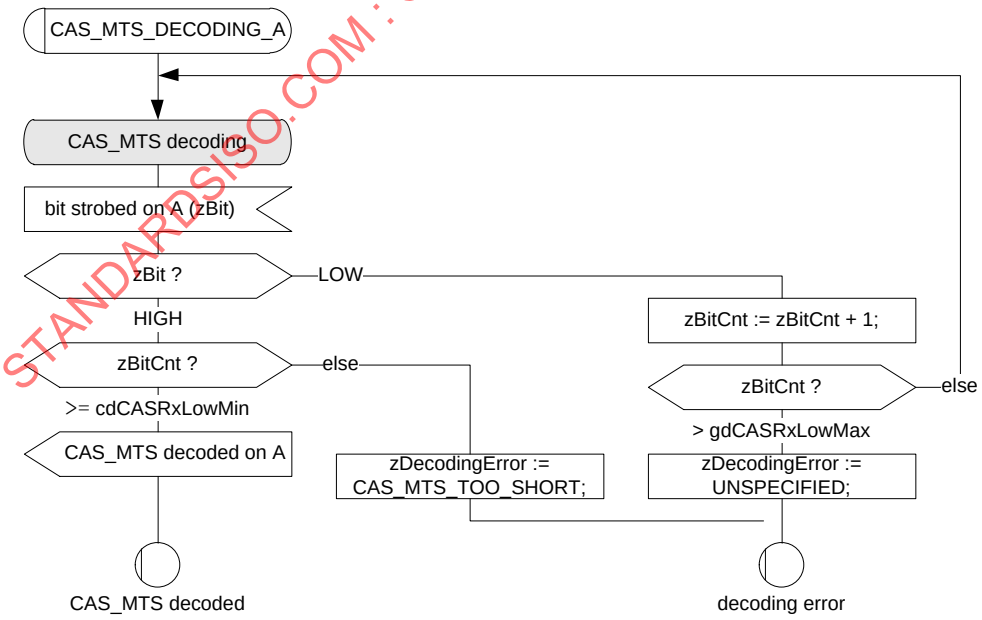



**Figure 65 — Decoding macro CAS_MTS_DECODING_A [CODEC_A]**

Figure 66 depicts the macro FSS_BSS_DECODING_A [CODEC_A].

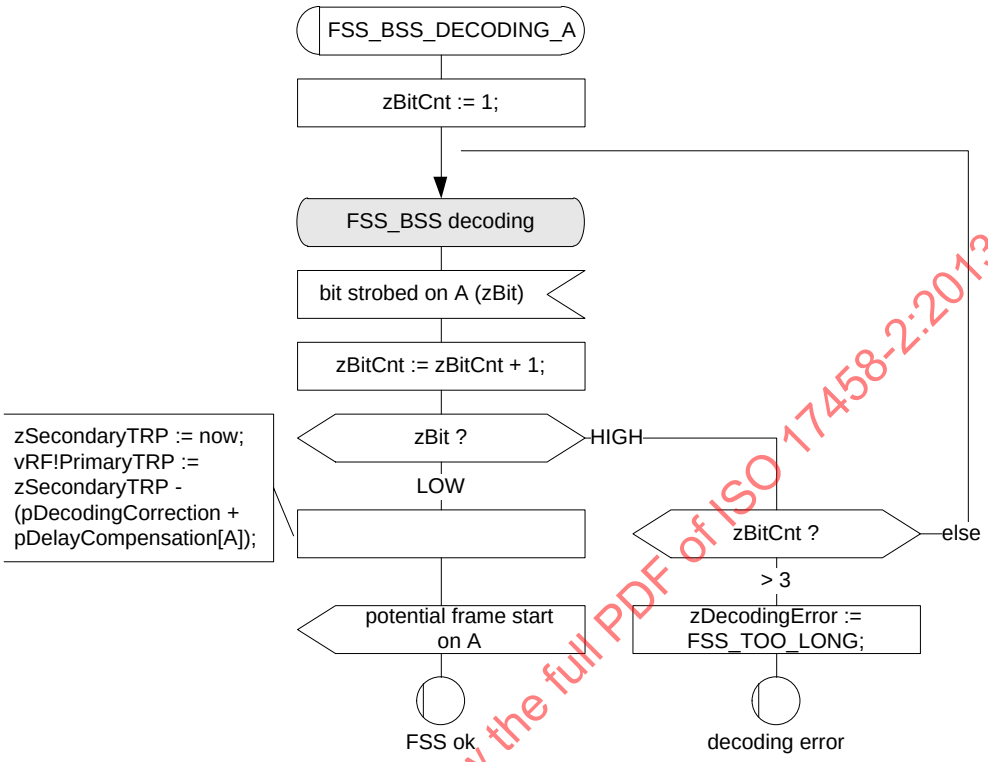



**Figure 66 — Decoding macro FSS_BSS_DECODING_A [CODEC_A][46)]**



46) If a bit is strobed at a microtick boundary 'now' should reflect the larger microtick value.

Figure 67 depicts the macro HEADER_DECODING_A [CODEC_A].

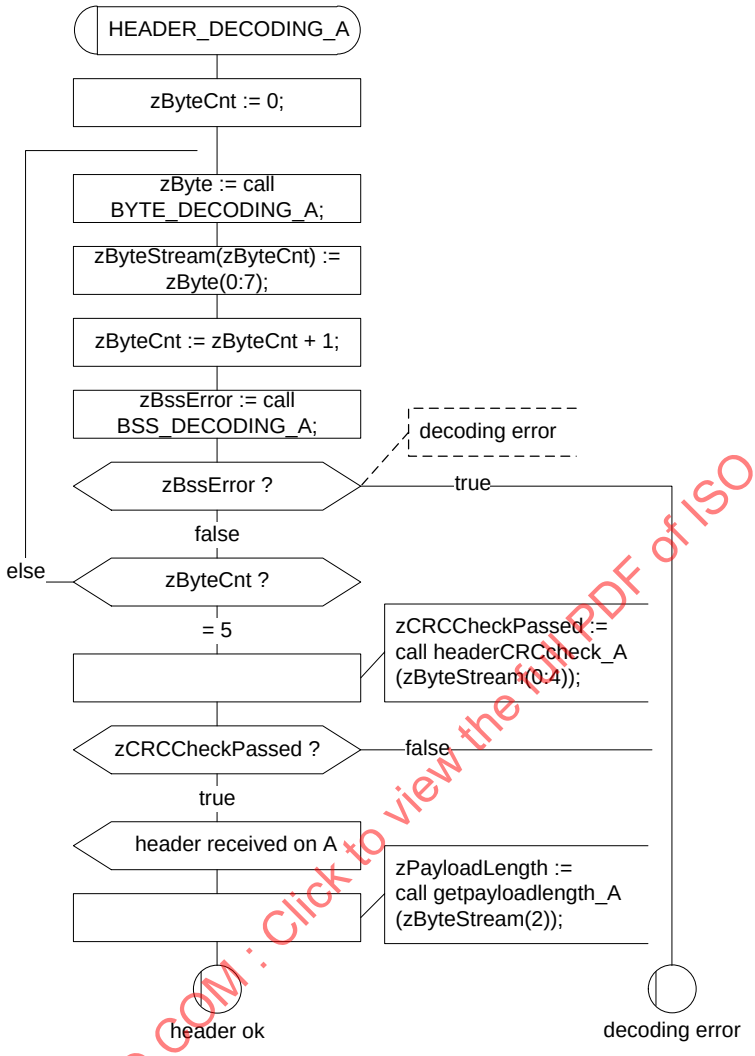



**Figure 67 — Decoding macro HEADER_DECODING_A [CODEC_A]**

The function **headerCRCcheck** returns a Boolean, *zCRCCheckPassed*, which is true if the header CRC check was passed (see 8.5.3) 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 71 and Figure 72.

Figure 68 depicts the procedure BYTE_DECODING_A [CODEC_A].

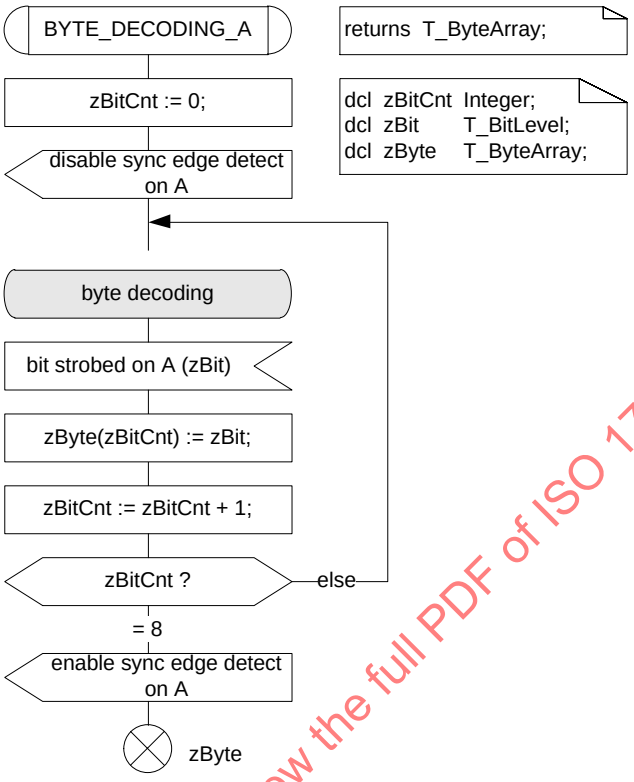



**Figure 68 — Procedure BYTE_DECODING_A [CODEC_A]**

Figure 69 depicts the procedure BSS_DECODING_A [CODEC_A].

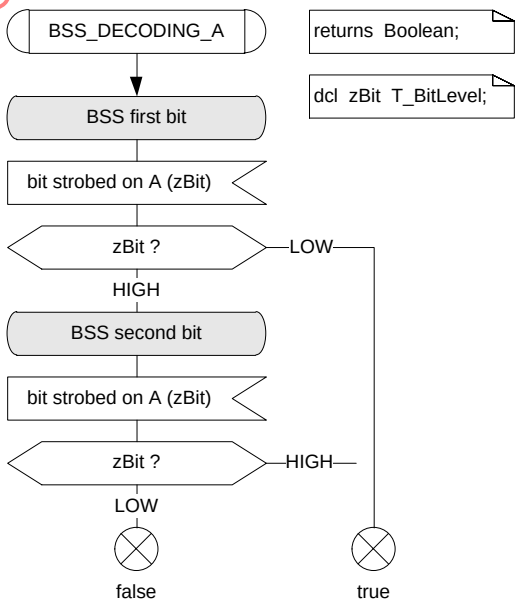



**Figure 69 — Procedure BSS_DECODING_A [CODEC_A]**

Figure 70 depicts the macro FES_DECODING_A [CODEC_A].

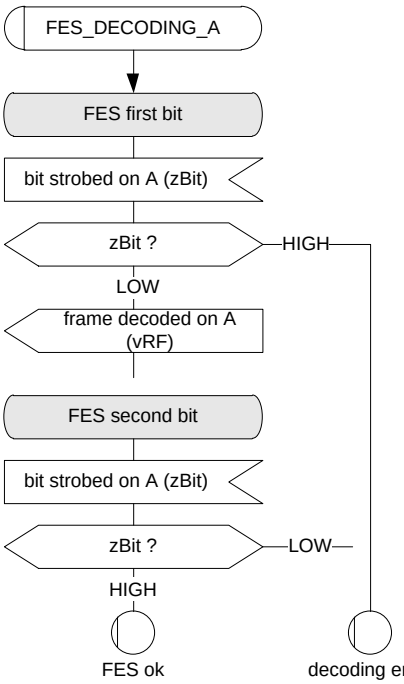



**Figure 70 — Decoding macro FES_DECODING_A [CODEC_A]**

Figure 71 depicts the macro PAYLOAD_DECODING_A [CODEC_A].

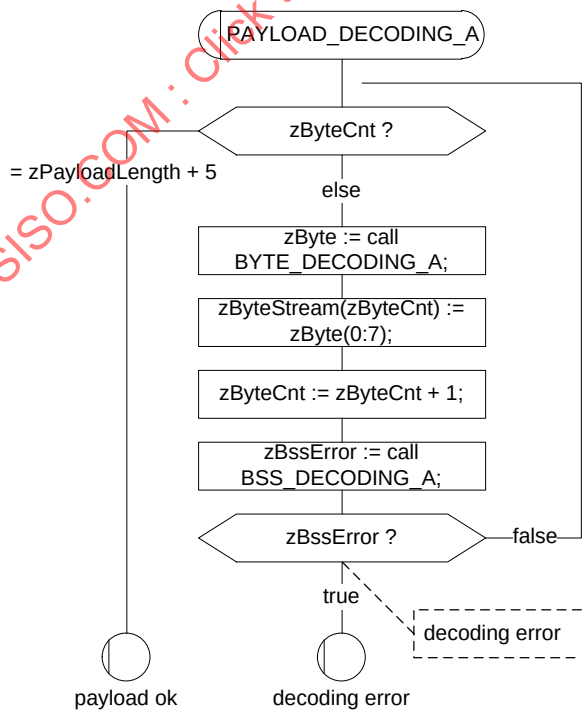



**Figure 71 — Decoding macro PAYLOAD_DECODING_A [CODEC_A]**

Figure 72 depicts the macro TRAILER_DECODING_A [CODEC_A].

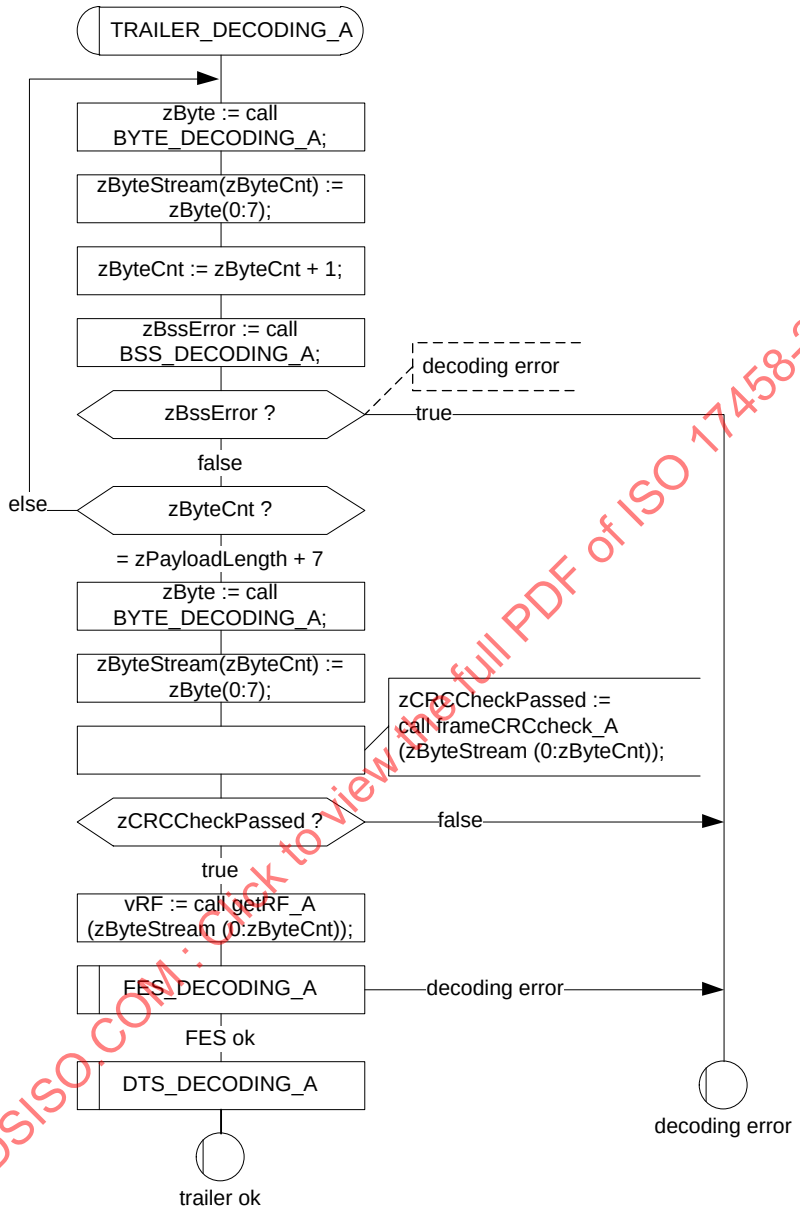



**Figure 72 — Decoding macro TRAILER_DECODING_A [CODEC_A]**



The function **frameCRCcheck** returns a Boolean, *zCRCCheckPassed*, which is true if the frame CRC check was passed (see 8.5.4) and is false if the frame CRC check fails. This function is channel specific due to the channel specific initialisation vectors of the CRC calculation (see 8.5.4 for details).

The function **getRF** used in Figure 72 extracts decoded header and payload data from *zByteStream* and returns it via the structure variable *vRF*.

Figure 73 depicts the macro DTS_DECODING_A [CODEC_A].

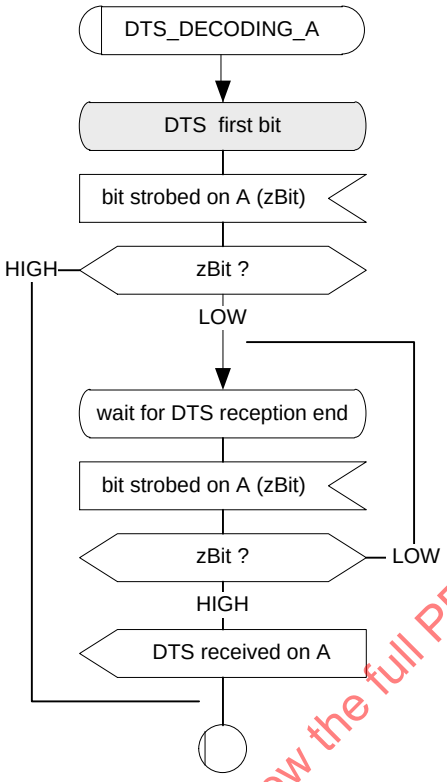



**Figure 73 — Decoding macro DTS_DECODING_A [CODEC_A]**



## 7.4 Bit strobing process

### 7.4.1 Operating modes

The receiving node shall strobe the received data from the BD according to the bit strobing process BITSTRB.

Definition **(20)** shows the formal definition of the BITSTRB operating modes:

**Definition: *T_StrbMode*** **(20)**

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

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

— In the STANDBY mode bit strobing is effectively halted.

— In the GO mode the bit strobing process shall be executed.

— In the BLIND mode the bit strobing process is paused for a configurable interval. After expiration of this interval the mode is switched automatically to GO.

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

### 7.4.2 Bit strobing process behaviour

Figure 74 depicts the BITSTRB process [BITSTRB_A].

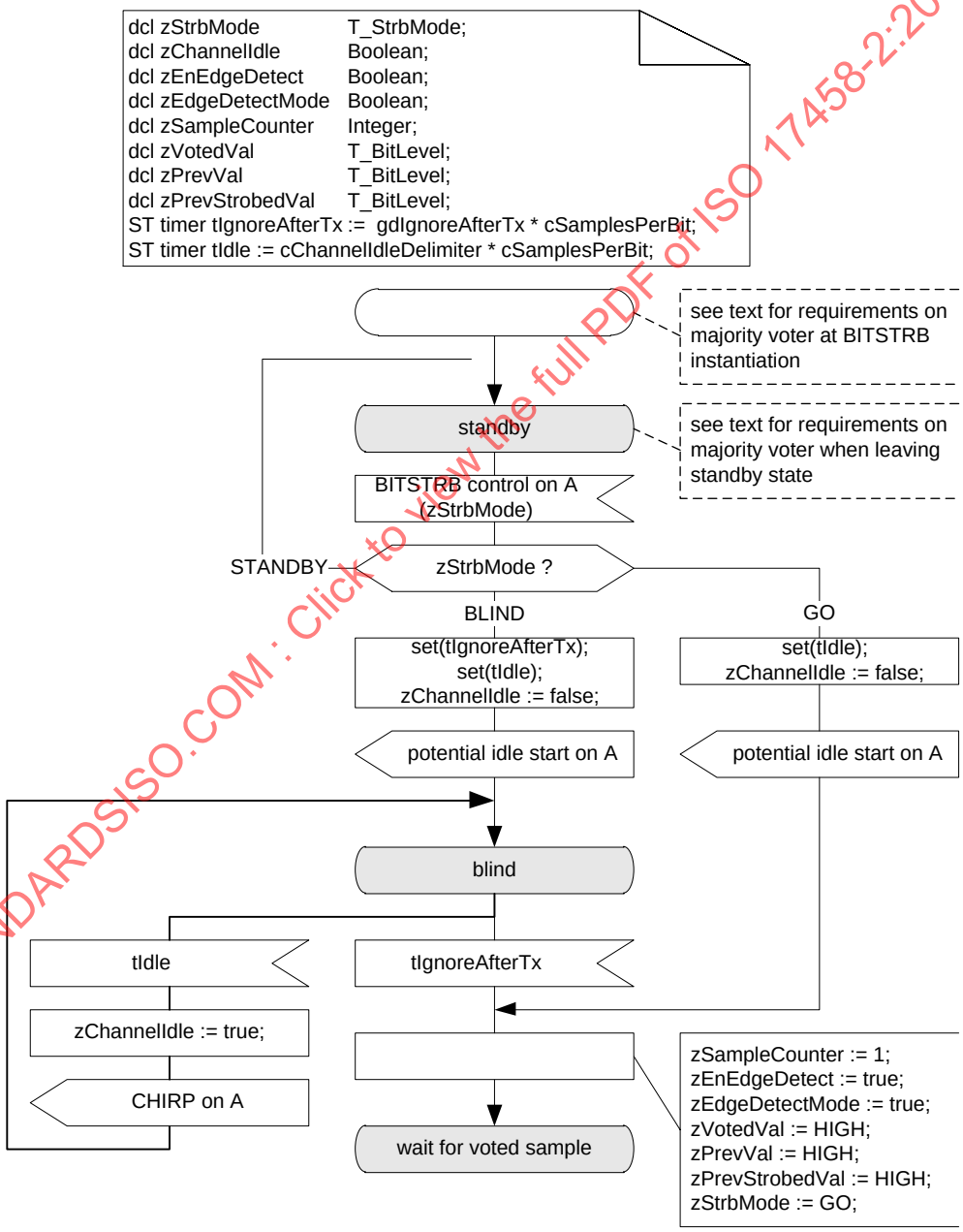



**Figure 74 — BITSTRB process [BITSTRB_A]**

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

On a transition out of the *BITSTRB:standby* state the majority voter shall behave as if it had been continuously sampling during the entire standby state[47)].

Figure 75 depicts the state "wait for voted sample" and the handling of input signals received in this state.

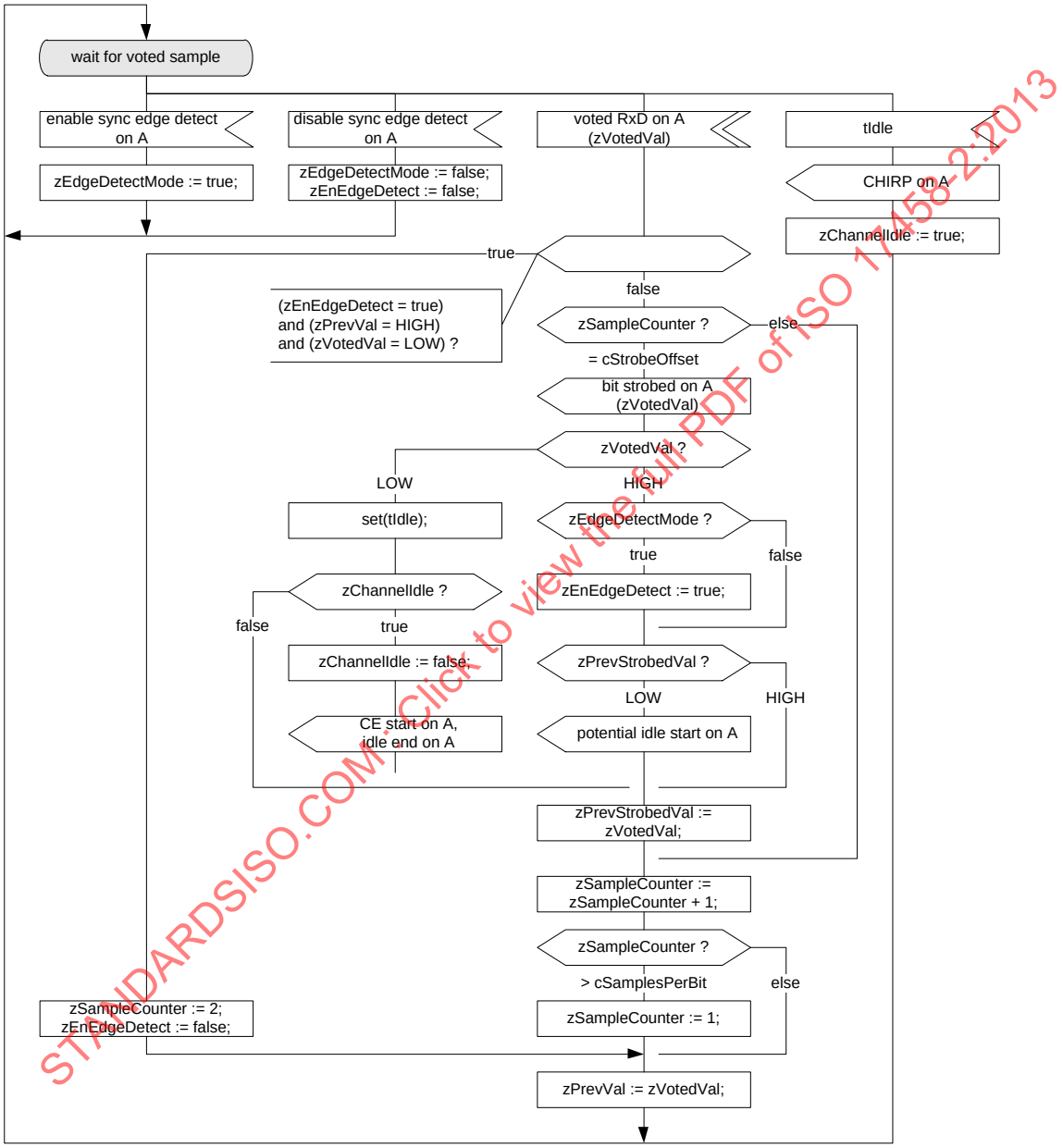



**Figure 75 — Wait for voted sample [BITSTRB_A]**

47) This is not a requirement that an implementation shall actually sample when BITSTRB is in the *BITSTRB:standby* state - an implementation that restarted sampling several samples before the transition out of standby would also be acceptable.

Figure 76 depicts the BITSTRB process control and process termination [BITSTRB_A].

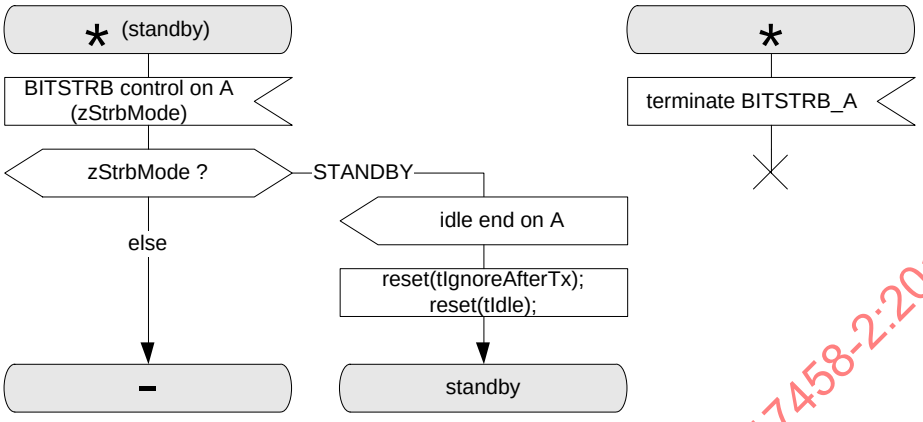



**Figure 76 — BITSTRB process control and process termination [BITSTRB_A]**

## 7.5 Wakeup pattern decoding process

### 7.5.1 Operating modes

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

The wakeup pattern decoding process WUPDEC distinguishes between two modes.

Definition **(21)** shows the WUPDEC operating modes:

**Definition: *T_WupDecMode*** **(21)**

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

— In the STANDBY mode, the wakeup pattern decoding process (WUPDEC) is effectively halted.

— In the GO mode, the node shall decode wakeup patterns.

### 7.5.2 Wakeup pattern decoding process behaviour

Figure 77 depicts the control of the wakeup pattern decoding process and its termination [WUPDEC_A].

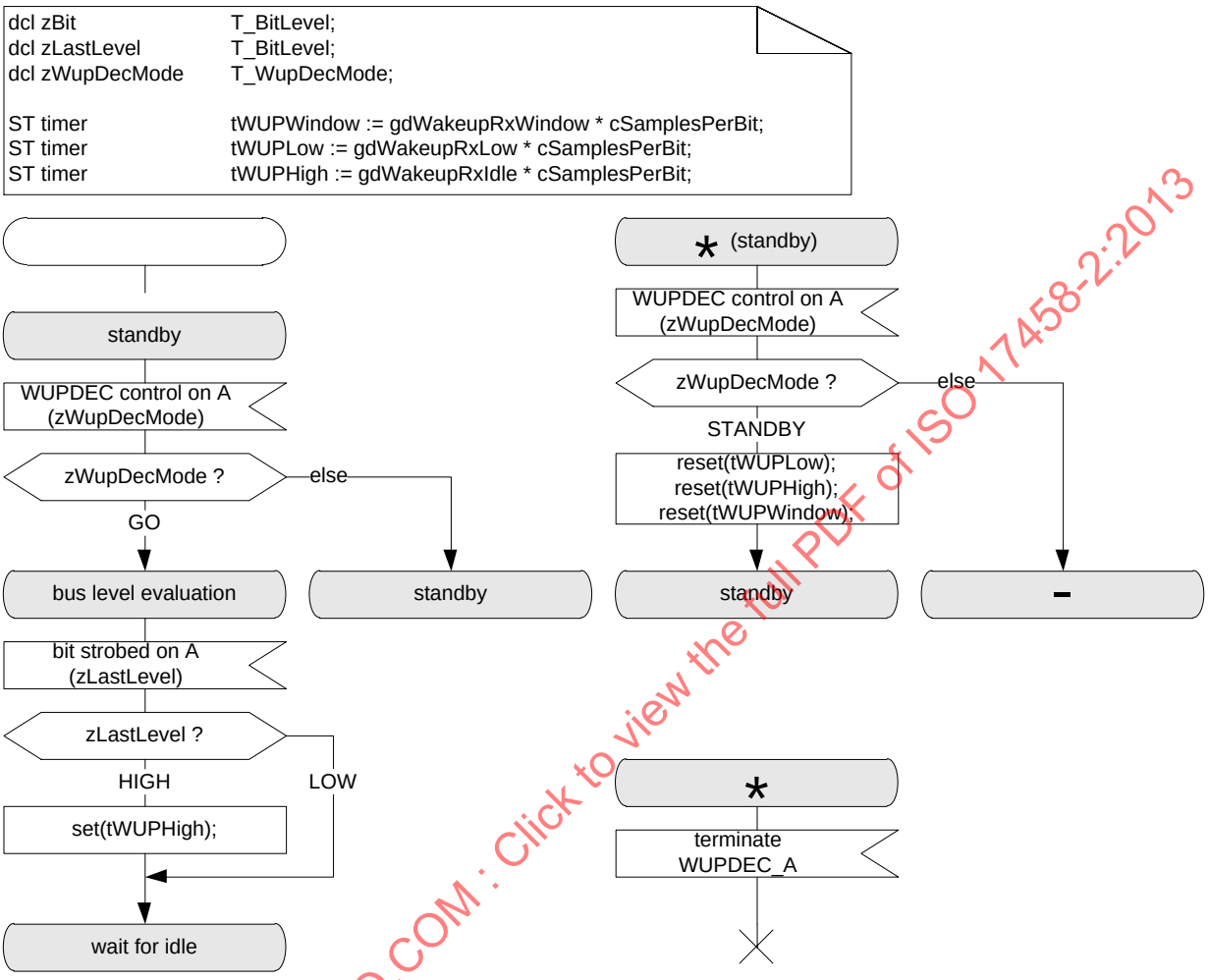



**Figure 77 — Control of the wakeup pattern decoding process and its termination [WUPDEC_A]**

Figure 78 depicts the Control of the wakeup timer [WUPDEC_A].

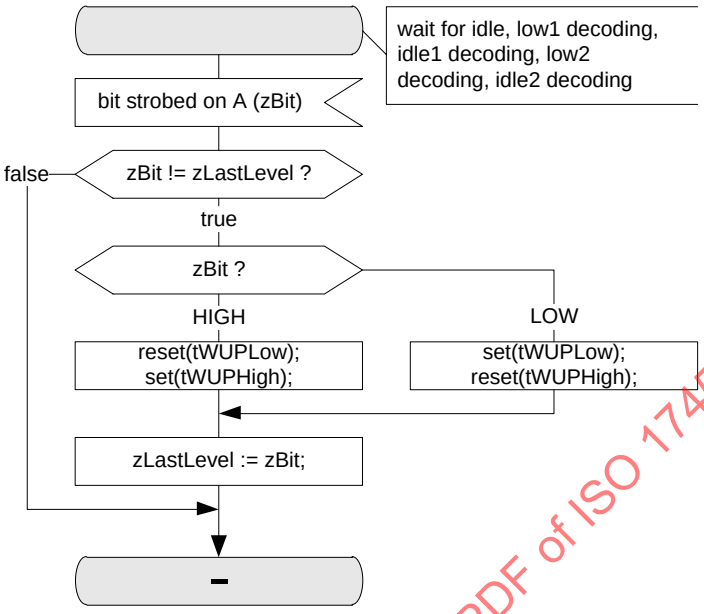



**Figure 78 — Control of the wakeup timer [WUPDEC_A]**

Figure 79 depicts the Wakeup pattern decoding [WUPDEC_A].

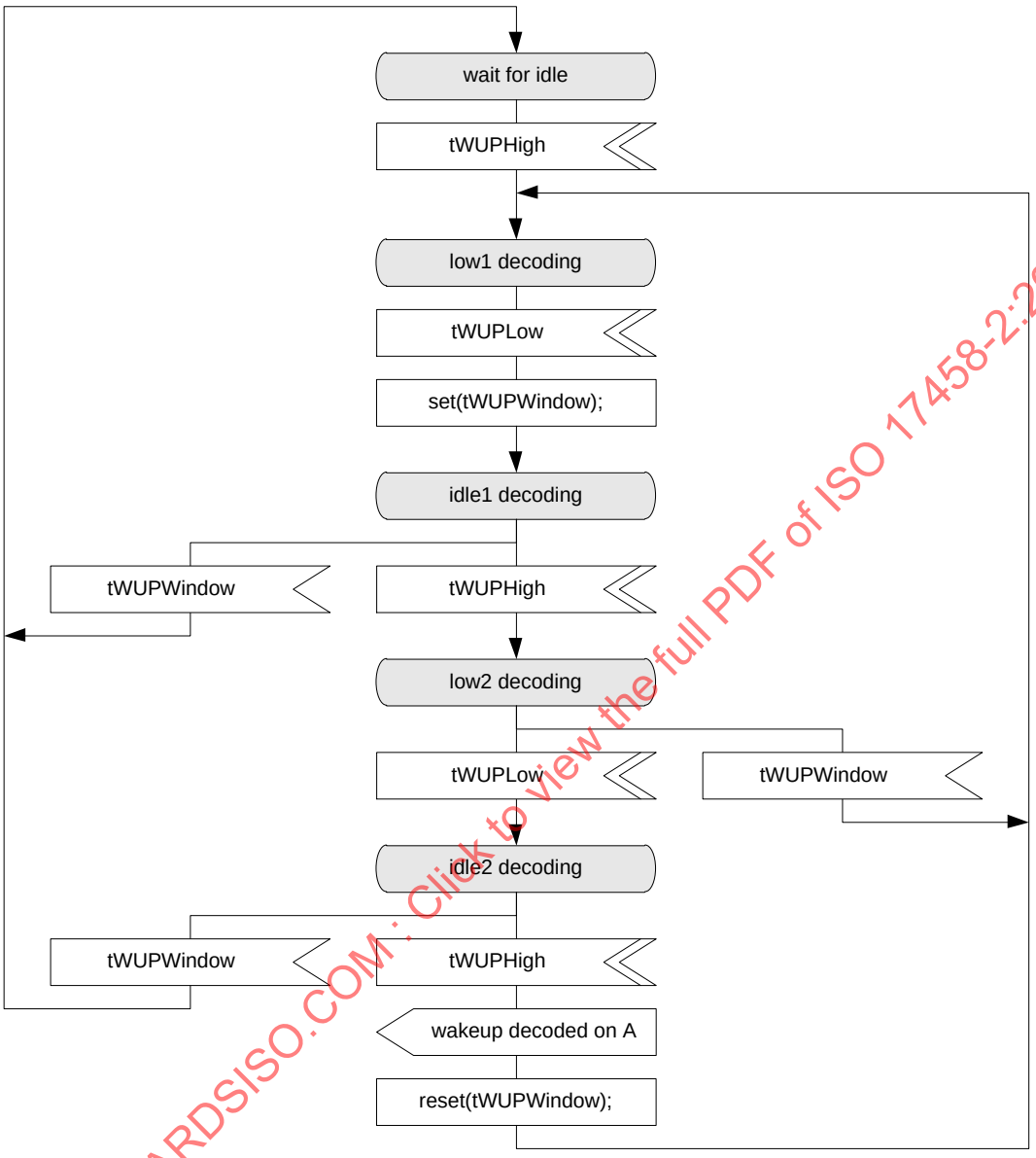



**Figure 79 — Wakeup pattern decoding [WUPDEC_A]**

# 8 Frame Format

## 8.1 Overview

An overview of the FlexRay frame format is given in Figure 80. The frame consists of three segments. These are the header segment, the payload segment, and the trailer segment.

Figure 80 depicts the FlexRay frame format.

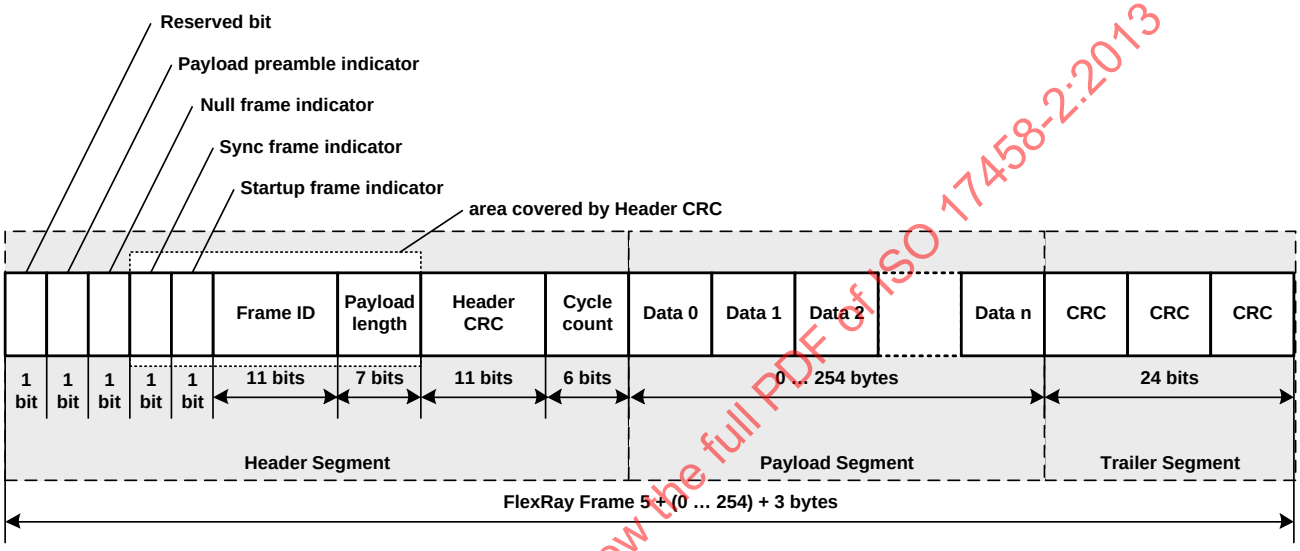



**Figure 80 — FlexRay frame format**

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 80, (in the header segment, for example, the reserved bit is transmitted first and the cycle count field is transmitted last).

## 8.2 FlexRay header segment (5 bytes)

### 8.2.1 General

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.

### 8.2.2 Reserved bit (1 bit)

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

— A transmitting node shall set the reserved bit to logical '0'.

— A receiving node shall ignore the reserved bit[48)].

**Definition: *T_Reserved*** **(22)**

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

### 8.2.3 Payload preamble indicator (1 bit)

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

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

— If the frame is transmitted in the dynamic segment the payload preamble indicator indicates the presence of a message ID at the beginning of the payload.

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

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

**Definition: *T_PPIndicator*** **(23)**

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

### 8.2.4 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[49)]. The conditions under which a transmitting node may

48) 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).

49) 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 shall 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.

send a null frame are detailed in clause 9. Nodes that receive a null frame may still use some information related to the frame[50)].

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

— If the null frame indicator is set to one then the payload segment contains data.

**Definition: *T_NFIndicator*** **(24)**

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

### 8.2.5 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 synchronisation of communication[51)].

— If the sync frame indicator is set to zero then no receiving node shall consider the frame for synchronisation or synchronisation related tasks.

— If the sync frame indicator is set to one then all receiving nodes shall utilize the frame for synchronisation if it meets other acceptance criteria (see below).

The clock synchronisation mechanism (described in clause 12) 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 synchronisation mechanism. Details of how the node shall set the sync frame indicator are specified in clause 9 and 12.8.

**Definition: *T_SyFIndicator*** **(25)**

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

### 8.2.6 Startup frame indicator (1 bit)

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

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

— If the startup frame indicator is set to one then the frame is a startup frame.

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

50) For example, the clock synchronisation 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).
51) Sync frames are only sent in the static segment. Please refer to the rules to configure sync frames.

The startup (described in clause 11) and clock synchronisation (described in clause 12) 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 clause 9 [52)].

**Definition: *T_SuFIndicator*** **(26)**

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

### 8.2.7 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 2 047[53)]. The frame ID 0 is an invalid frame ID[54)]. 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.

**Definition: *T_FrameID*** **(27)**

```
syntype T_FrameID = Integer
   constants 0 : 2047 55)
endsyntype;
```

### 8.2.8 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)[56)].

The payload length ranges from 0 to *cPayloadLengthMax*[57)] which corresponds to a payload segment containing 2 * *cPayloadLengthMax* bytes.

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

---

52) 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.

53) In binary: from $(000\ 0000\ 0001)_2$ to $(111\ 1111\ 1111)_2$.

54) The frame ID of a transmitted frame is determined by the value of *vSlotCounter(Ch)* at the time of transmission (see clause 13). 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.

55) Frame IDs range from 1 to 2 047. The zero is used to mark invalid frames, empty slots, etc.

56) The payload length field does not include the number of bytes within the header and the trailer segments of the FlexRay frame.

57) The electrical physical layer puts a restriction on the usable payload such that the overall transmission duration is limited (see ISO 17458-4. It may restrict the payload length at 2,5 and 5 Mbit / s. For details please refer to B.4.41 and B.4.42.

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.

**Definition: *T_Length*** **(28)**

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

### 8.2.9 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)[58]. 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[59] shall be

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

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

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

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 8.5.3.

**Definition: *T_HeaderCRC*** **(29)**

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



58) 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).

59) 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).

### 8.2.10 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 subclauses 9.3.3.2 and 9.3.4.2).

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.

**Definition: *T_CycleCounter*** **(30)**

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

### 8.2.11 Formal header definition

The formal definitions of the fields in the previous subclauses and the header segment structure depicted in Figure 80 yield the following formal definition for the header segment:

**Definition: *T_Header*** **(31)**

```
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;
```

## 8.3 FlexRay payload segment (0 – 254 bytes)

### 8.3.1 Payload

The FlexRay payload segment contains 0 to 254[60)] bytes (0 to 127 two-byte words) of data. Because the payload length contains the number of two-byte words, the payload segment contains an even number of bytes. The bytes of the payload segment are identified numerically, starting at 0 for the first byte after the header segment and increasing by one with each subsequent byte. The individual bytes are referred to as "Data 0", "Data 1", "Data 2", etc., with "Data 0" being the first byte of the payload segment, "Data 1" being the second byte, etc.

The frame CRC described in subclause 8.5.4 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.

60) The electrical physical layer puts a restriction on the usable payload such that the overall transmission duration is limited (see ISO 17458-4). It may restrict the payload length at 2,5 and 5 Mbit / s. For details please refer to B.4.41 and B.4.42.

The payload preamble indicator in the frame header indicates whether the payload segment contains the message ID.

For frames transmitted in the static segment the first 0 to 12 bytes of the payload segment may be used as a network management vector. The payload preamble indicator in the frame header indicates whether the payload segment contains the network management vector[61]. 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[62].

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.

**Definition: *T_Payload*** **(32)**

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

### 8.3.2 NMVector

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

— The length of the NMVector is configured during *POC:config* by the parameter *gNetworkManagementVectorLength*. All nodes in a cluster shall 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 subclause 13.3.3.4.

— 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[63].

Figure 81 depicts the payload segment of frames transmitted in the static segment.



61) 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.
62) 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.
63) This allows lower bits to remain at defined positions if the length of the NMVector changes.

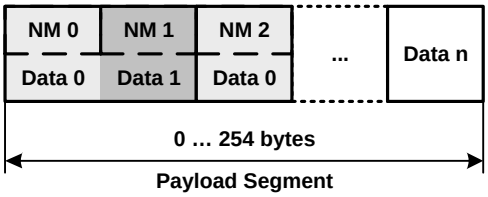



**Figure 81 — Payload segment of frames transmitted in the static segment**

### 8.3.3 Message ID (16 bits)

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

— The message ID is an application determined number that identifies the contents of the data segment.

— The message ID can only be used in frames transmitted in the dynamic segment.

— The message ID is 16 bits long.

— At the transmitting node the message ID is written by the host as application data. The protocol engine has no knowledge about the message ID and no mechanism inside the protocol engine is based on the message ID.

— At the receiving node the storage of a frame may depend on the result of a filtering process that makes use of the message ID. All frame checks done in Frame Processing (see clause 10) are unmodified (i.e., are not a function of the message ID). The use of the message ID filter is defined in subclause 13.3.2.11.3.6.

— The presence or absence of a message ID is indicated in the frame header by the payload preamble indicator.

— If this mechanism is used, the most significant bit of the MessageID shall be placed in the most significant bit of the first byte of the payload segment. Subsequent bits of the message ID shall be placed in the next payload bits in order of decreasing significance.

Figure 82 depicts the payload segment of frames transmitted in the dynamic segment.

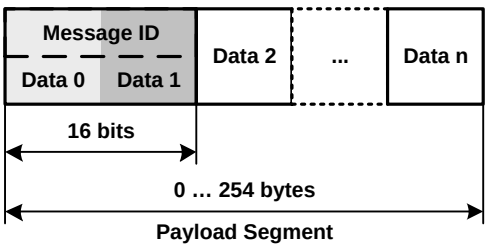



**Figure 82 — Payload segment of frames transmitted in the dynamic segment**

## 8.4 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[64].

The CRC is computed using the same generator polynomial on both channels. The CRC polynomial[65] shall be

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

The node shall use a different initialisation vector depending on which channel the frame should be transmitted[66]

— The node shall use the initialisation vector 0xFEDCBA for frames sent on channel A.

— The node shall use the initialisation 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 subclause 8.5.4.

**Definition: *T_FrameCRC*** **(33)**

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

## 8.5 CRC calculation details

### 8.5.1 Context of the CRC calculation

The behaviour 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 67 and Figure 72). The CODEC also appends the CRC in the trailer segment of a transmitted frame (see subclause 7.2.1.2.6). The algorithm executed to calculate the CRC is the same in all cases except for the initial values of several algorithm parameters.

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

65) This 24-bit CRC polynomial generates a code that has a minimum Hamming distance of 6 for codewords up to 2 048 bits in length and a minimum Hamming distance of 4 for codewords up to 4 094 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.

66) Different initialisation 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).

### 8.5.2 CRC calculation algorithm

Initialize the CRC shift register with the appropriate initialisation 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[67] 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)
   // 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
```

### 8.5.3 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[68].

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 subclause 8.5.2 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) initialisation vector of header
                               // CRC for both channels
```

67) Transmitting nodes use the bit sequence that will be fed into the coding algorithm (see clause 7), 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.)).

68) 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.

```
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.

#### 8.5.4 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 8.5.2 is used to calculate the header CRC. The parameters for the algorithm are defined as follows:

**FlexRay frame CRC calculation algorithm parameters - channel A:**

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

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

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

# 9 Media Access Control

## 9.1 Principles

### 9.1.1 Overview

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.

### 9.1.2 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 83.

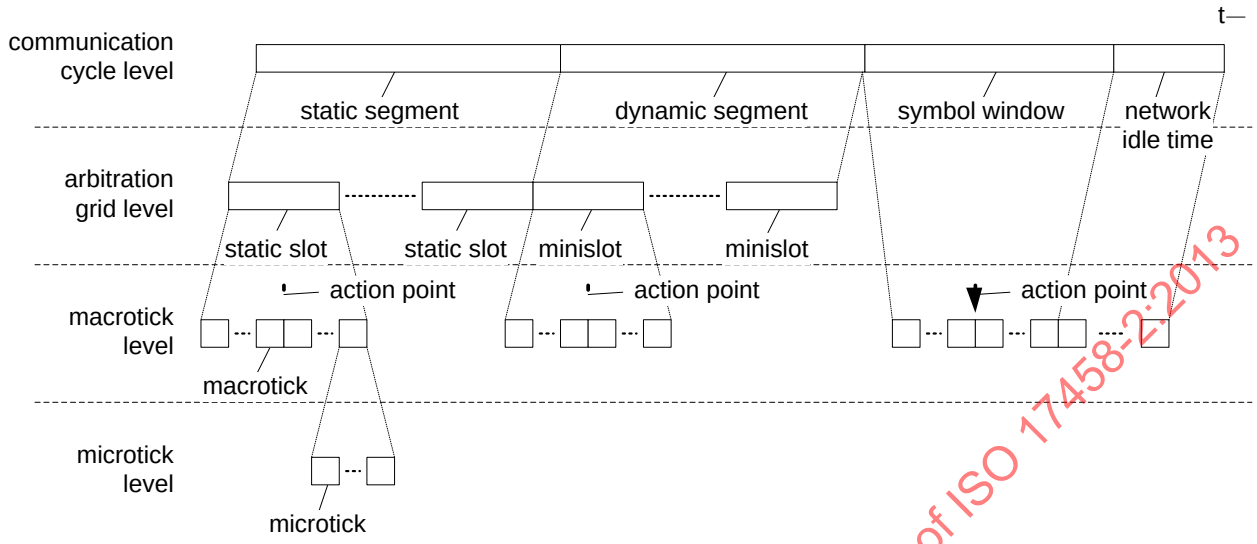



**Figure 83 — Timing hierarchy within the communication cycle**

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 9.3.3. Within the dynamic segment a dynamic mini-slotting based scheme is used to arbitrate transmissions as specified in 9.3.4. The symbol window is a communication period in which a symbol can be transmitted on the network as specified in 9.3.5. The network idle time is a communication-free period that concludes each communication cycle as specified in 9.3.6.

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 clause 12. 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 clause 12.

### 9.1.3 Communication cycle execution

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

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

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

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

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

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

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

Figure 84 illustrates the overall execution of the communication cycle.

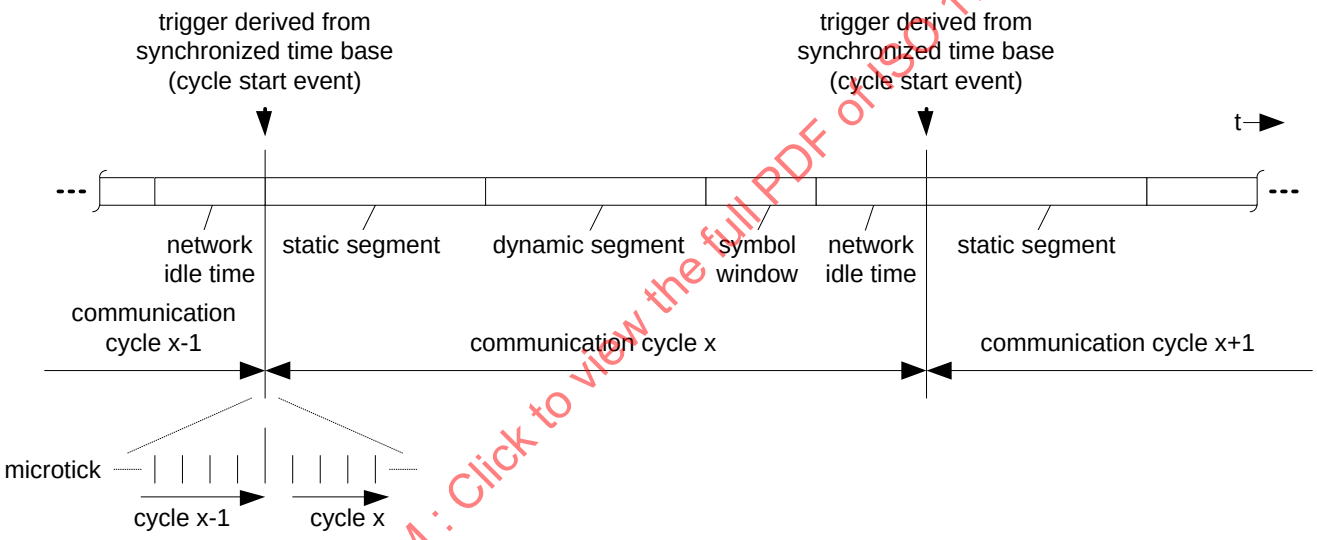



**Figure 84 — Time base triggered communication cycle**

The node shall maintain a cycle counter *vCycleCounter* that contains the number of the current communication cycle. Initialisation and maintenance of the cycle counter are specified in clause 12.

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

### 9.1.4 Static segment

#### 9.1.4.1 Structure of the static segment

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

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.

— If a node has a key slot or key slots, that node shall transmit a frame in the key slot(s) on all connected channels in all communication cycles.

— In slots other than the key slot(s), frames may be transmitted on either channel, or on both.

— In a given communication cycle, no more than one node shall transmit a frame with a given frame ID on a given channel. It is allowed, however, for different nodes to transmit frames with the same frame ID on the same channel in different communication cycles.

— If a non-sync node is configured to enter key slot mode after startup (i.e., *pKeySlotOnlyEnabled* is true) the node shall designate one frame as the key slot frame via the parameter *pKeySlotID*.

#### 9.1.4.2 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 85 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[69]. In slot 3 no frame is transmitted on either channel.

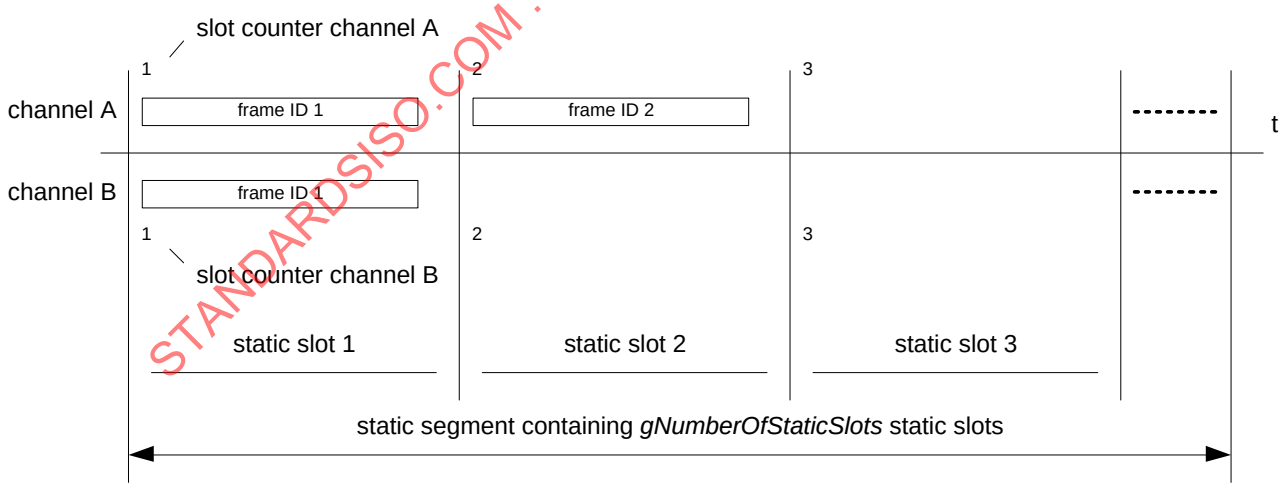



**Figure 85 — Structure of the static segment**

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

69) 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.

For any given node one or two[70] static slots (as defined in *pKeySlotID* and, in some cases, *pSecondKeySlotID*) may be assigned to contain sync frames (as identified by *pKeySlotUsedForSync*), a special type of frame required for synchronisation within the cluster. Specific sync frames may be assigned to be startup frames (as identified by *pKeySlotUsedForStartup*).

Figure 86 depicts the timing within the static segment.

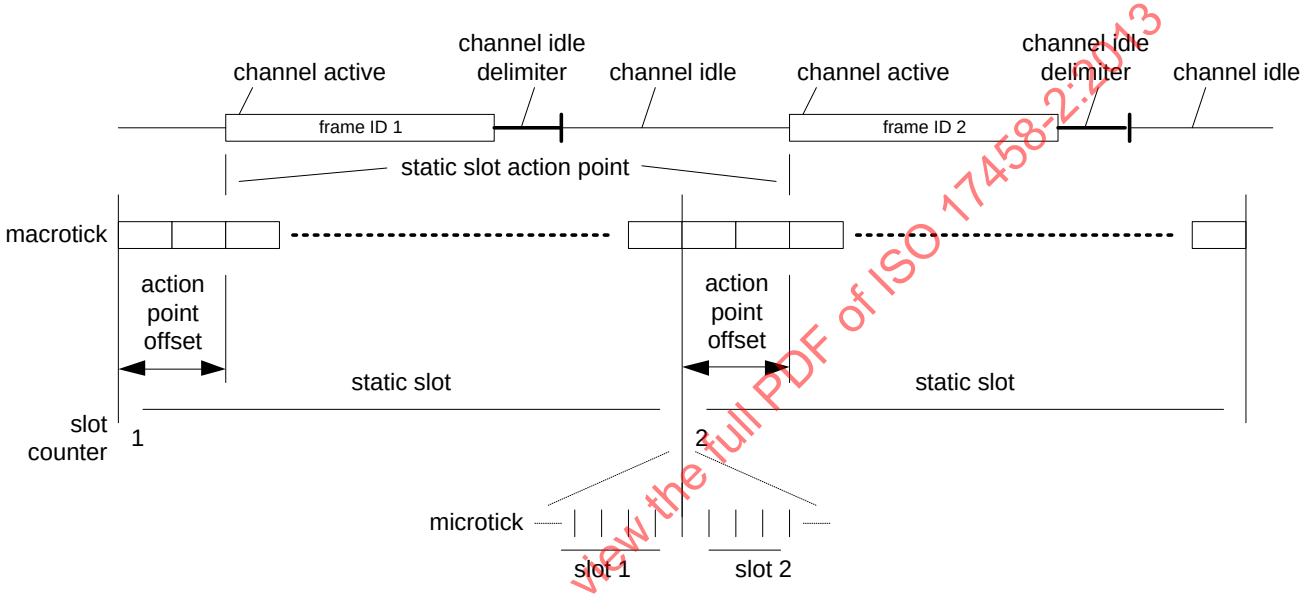



**Figure 86 — Timing within the static segment**

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.

### 9.1.5 Dynamic segment

#### 9.1.5.1 Structure of the dynamic segment

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

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

For communication within the dynamic segment the following constraints apply.

— Sync frames, startup frames, and null frames are not allowed.

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

— In a given communication cycle, transmission of a given frame may be attempted on either channel, on both channels, or on neither channel[71]. Due to the independent nature of the channel-dependent media access processes it is possible that frame transmission that is attempted on both channels may be successful on one channel but unsuccessful on the other. It is also possible that a frame successfully transmitted on both channels is transmitted at different points in time on the different channels.

— In a given communication cycle, no more than one node shall transmit a frame with a given frame ID on a given channel. It is allowed, however, for different nodes to transmit frames with the same frame ID on the same channel in different communication cycles.

— A node cannot transmit frames in the dynamic segment when operating in key slot only mode. As a consequence, a node's key slot frames (as determined by the parameter *pKeySlotID* or *pSecondKeySlotID*) cannot be sent in the dynamic segment.

#### 9.1.5.2 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 87 outlines the media access scheme within the dynamic segment. As illustrated in Figure 87, 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.

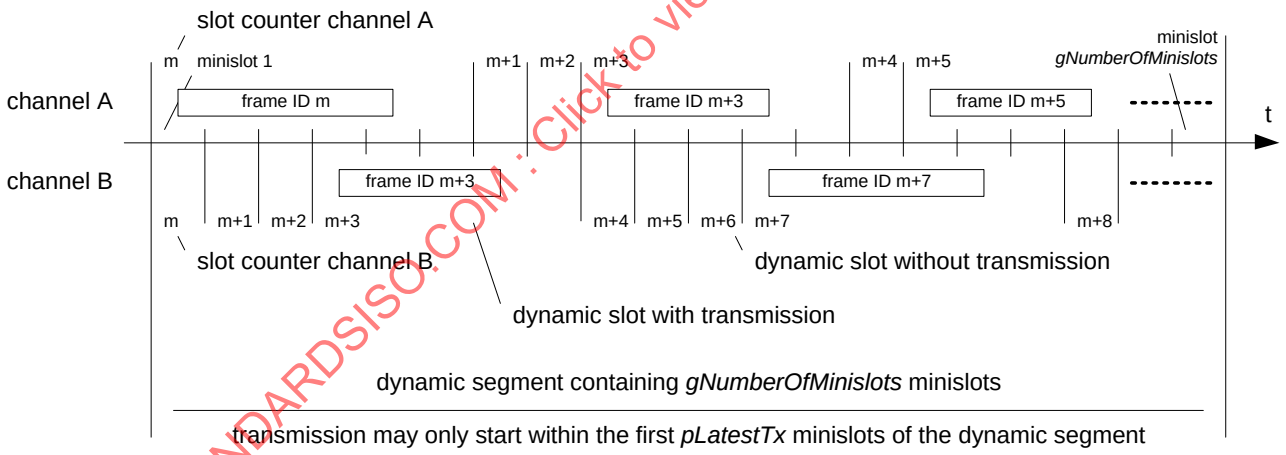



**Figure 87 — Structure of the dynamic segment**

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

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

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 87 illustrates how the duration of a dynamic slot adapts depending on whether or not communication takes place.

The node performs slot counting in the following way.

— The dynamic slot consists of one minislot if no activity takes place on the channel, i.e. the communication channel is in the channel idle state throughout the corresponding minislot.

— The dynamic slot consists of one or more minislots if activity takes place on the channel. If this activity is a legitimate frame transmission (as opposed to noise) the slot will consist of at least two minislots.

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

Figure 88 depicts the detailed timing of a minislot.

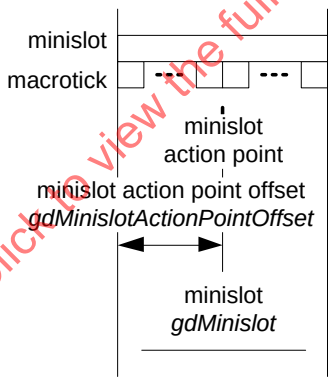



**Figure 88 — Timing within a minislot**

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 clause 7.

In contrast to a static slot, the dynamic slot consists of two distinct phases - a mandatory dynamic slot transmission phase and an optional dynamic slot idle phase.

The dynamic slot transmission phase extends from the start of the dynamic slot to the end of the last minislot in which the transmission terminates. The dynamic slot idle phase is an optional phase that extends from the end of the dynamic slot transmission phase to the end of the dynamic slot. Both the dynamic slot transmission phase and the dynamic slot idle phase (if it exists) consist of an integral number of minislots.

The optional dynamic slot idle phase is defined as a communication-free phase that provides additional time to allow all nodes to complete idle detection within the dynamic slot. A dynamic slot idle phase is not always required - in some cases the time difference between the point at which transmission ends (which always

occurs at a minislot action point) and the end of the minislot is sufficient to allow all nodes in the system to detect idle - such systems do not require a dynamic slot idle phase.

In some cases, however, the time after the minislot action point would be insufficient to ensure that all nodes in the system can detect idle within the dynamic slot. In such systems, every dynamic slot actually used for transmission is extended by a dynamic slot idle phase, allowing all nodes to detect idle within the dynamic slot.

Figure 89 depicts the structure of a dynamic slot that makes use of a dynamic slot idle phase.

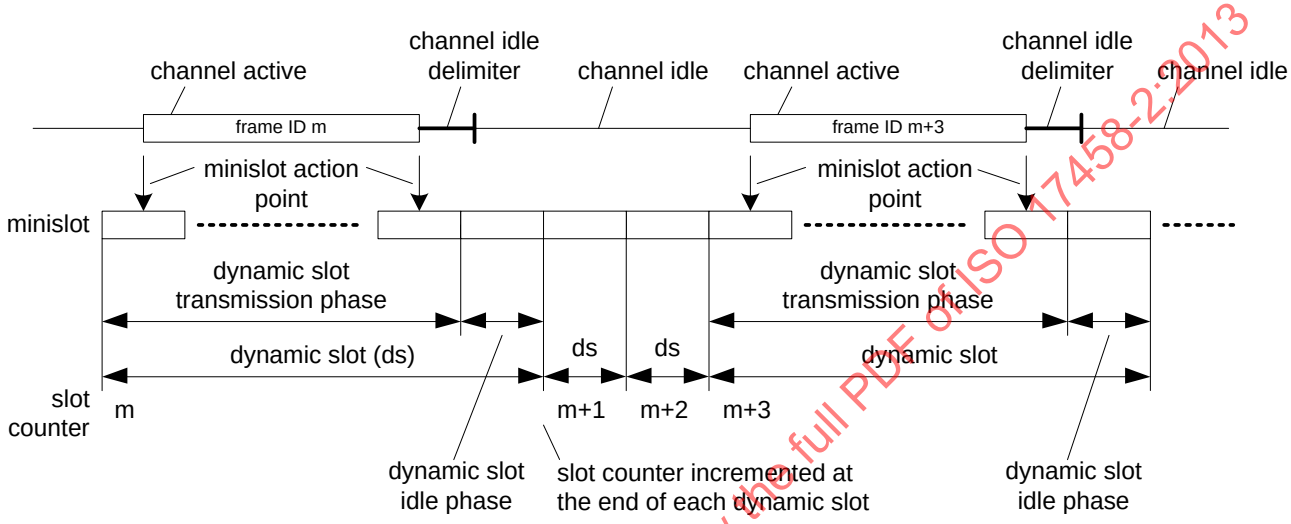



**Figure 89 — Structure of dynamic slots**

The start of the dynamic segment requires particular attention. The first action point in the dynamic segment occurs *gdActionPointOffset* macroticks after the end of the static segment if *gdActionPointOffset* is larger than *gdMinislotActionPointOffset* else it occurs *gdMinislotActionPointOffset* macroticks after the end of the static segment[72)].



72) 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.

The two cases are illustrated in Figure 90.

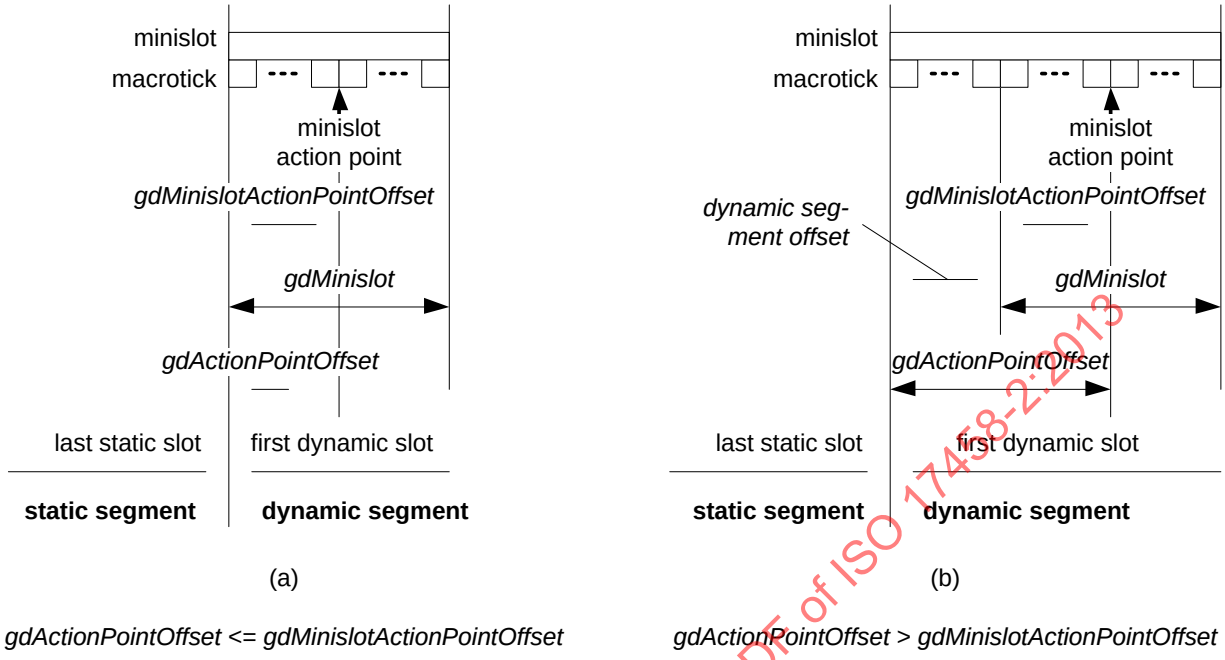



**Figure 90 — Timing at the boundary between the static and dynamic segments**

The node performs slot counter housekeeping on a per channel basis. At the end of every dynamic slot the node generally increments the slot counter *vSlotCounter* by one[73]. This is done until either

— the channel's slot counter has reached *cSlotIDMax*, or

— the dynamic segment has reached the minislot *gNumberOfMinislots,* i.e. the end of the dynamic segment.

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

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

The arbitration of the dynamic segment relies heavily on the assumption that all *CE start*, *potential idle start* and *CHIRP* signals are generated in response to legitimate frame transmissions by other nodes. Should disturbances on the physical layer make it through the majority voting, this may be violated. As a consequence, it may occur that different nodes in the cluster have different notions of the slot counter value for a given minislot.

Such a situation can also arise, for example, if the BD of a node is temporarily prevented from reception during a portion of the dynamic segment (i.e. by overtemperature or undervoltage conditions). Dynamic segment desynchronisation can also occur in certain situations where noise asymmetrically affects reception in the dynamic segment[74].



73) Under special circumstances, the node may increase the slot counter by two to prevent a desynchronisation of slot counters due to disturbances on the physical link.

74) This can occur in some situations even though the MAC process in the dynamic segment has several mechanisms to prevent desynchronisation when such noise exists.

When slot counter desynchronisation does occur in the dynamic segment its effect is limited to the specific instance of the dynamic segment. In other words, such desynchronisation will not affect the operation of the static segment or the overall clock synchronisation of the network, and will automatically be corrected at the end of the dynamic segment in which it occurred. The system designer should be aware of the possibility that slot counter desynchronisation could occur in the dynamic segment.

The MAC has the ability to correct certain fault scenarios where activity caused by noise on the physical layer is shorter than *cFrameThreshold* bits and also short enough that both the start of the activity and the end of the idle detection following the activity both occur in a single minislot or within two adjacent minislots.

If the MAC process detects such a noise event, it tries to revert to a state where it would have been if the noise event did not occur. This may cause an increment of the slot counter by two, or in some cases might even cause a slot counter increment during a subsequent reception (see Figure 103). To increase the robustness of the cluster, a node is prohibited from transmitting in the dynamic slot following a slot in which a noise event is detected.

### 9.1.6 Symbol window

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

Figure 91 outlines the media access scheme within the symbol window.

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

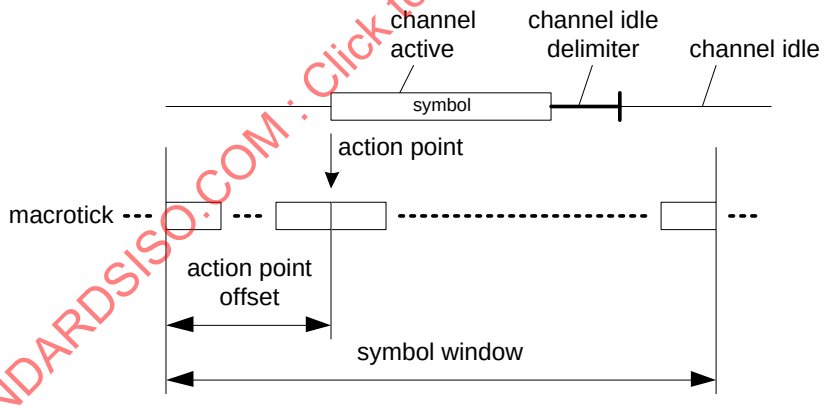



**Figure 91 — Timing within the symbol window**



### 9.1.7 Network idle time

The network idle time serves as a phase during which the node calculates and applies clock correction terms. Clock synchronisation is specified in clause 12.

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

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

## 9.2 Description

### 9.2.1 Relationship to other processes

The relationship between the Media Access Control processes and the other protocol processes is depicted in Figure 92[75)].

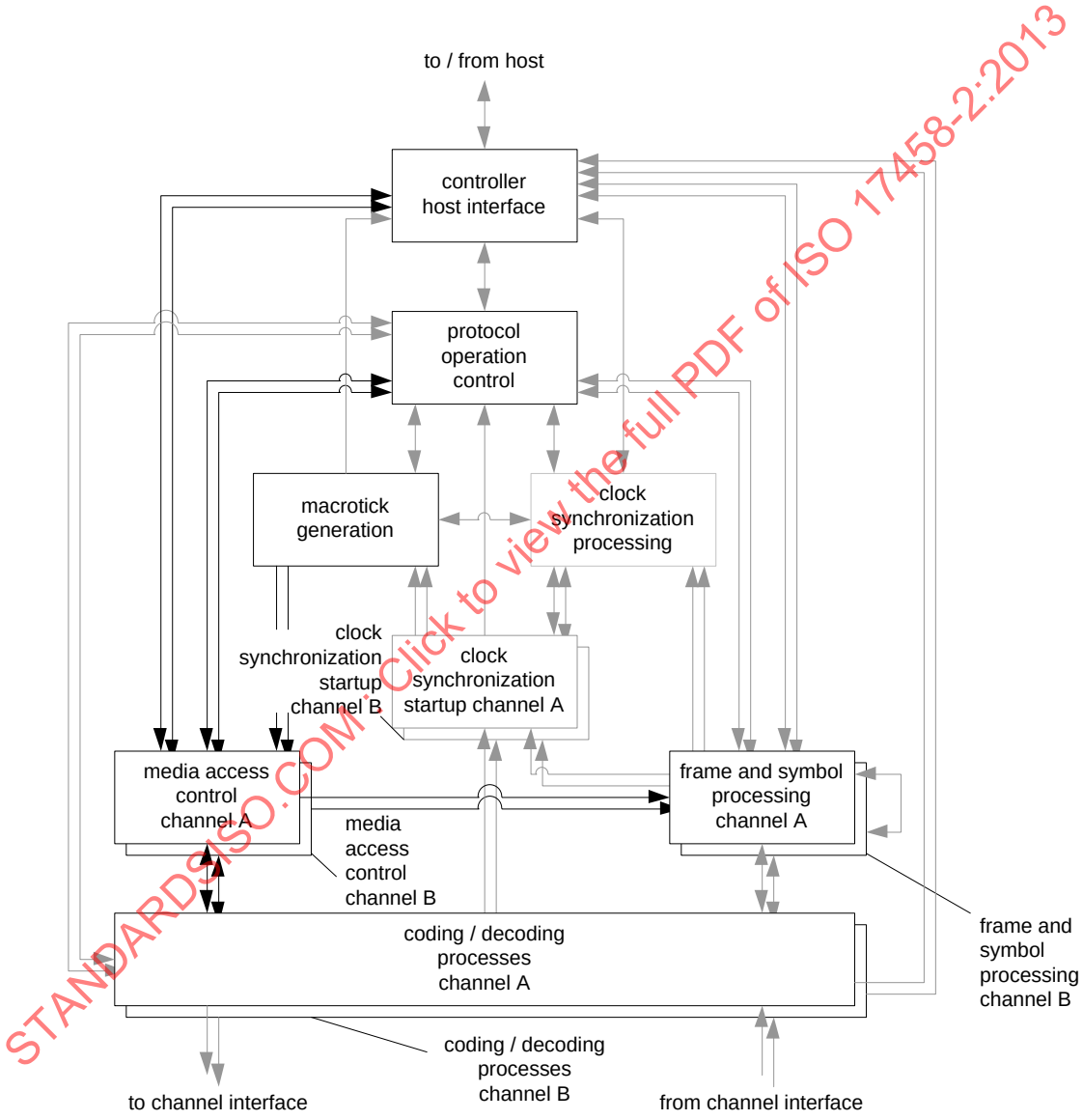





**Figure 92 — Media access control context**

75) The dark lines represent data flows between mechanisms that are relevant to this subclause. The lighter gray lines are relevant to the protocol, but not to this clause.

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.

### 9.2.2 Operating modes

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

— In the STANDBY mode media access is effectively halted.

— In the NOCE mode the media access process is executed, but no frames or symbols are sent on the channels.

— In the STARTUPFRAMECAS mode transmissions are restricted to the transmission of one startup null frame per cycle on each configured channel in each key slot if the node is configured to send a startup frame. In addition the node sends an initial CAS symbol prior to the first communication cycle.

— In the STARTUPFRAME mode transmissions are restricted to the transmission of one startup null frame per cycle on each configured channel in each key slot if the node is configured to send a startup frame.

— In the KEYSLOTONLY mode the transmissions are restricted based on the synchronisation type of the cluster and the role of the node. Sync nodes in a TT-D cluster have transmissions restricted to one sync frame per cycle on each configured channel. Coldstart nodes in a TT-E or TT-L cluster have transmissions restricted to two startup frames per cycle on each configured channel. Non-sync nodes in any cluster type have transmissions restricted to a single specified key slot frame[76)] per cycle on each configured channel.

— In the ALL mode frames and symbols are sent in accordance with the node's transmission slot allocation.

Definition **(34)** gives the formal definition of the MAC operating modes.

**Definition: *T_MacMode*** **(34)**

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

### 9.2.3 Significant events

#### 9.2.3.1 Event types

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.

#### 9.2.3.2 Reception-related events

Figure 93 depicts the reception-related events that are significant for media access control.

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



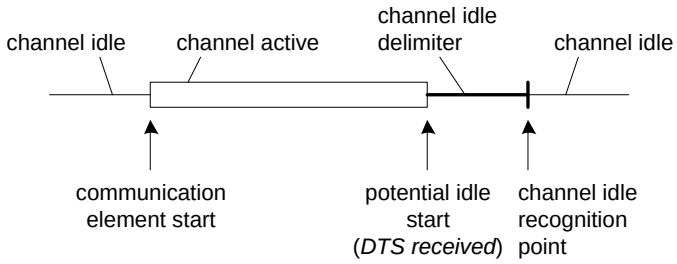



**Figure 93 — Reception-related events for MAC**

For communication channel A the reception-relevant events are

— communication element start on channel A (signal *CE start on A*, signal *idle end on A*),

— potential idle start on channel A (signal *potential idle start on A*),

— channel idle recognition point detected on channel A (signal *CHIRP on A*),

— DTS high bit received on channel A (signal *DTS received on A*, only in the dynamic segment), and

— bit strobed on channel A (signal *bit strobed on A*, not shown in Figure 93).

For communication channel B the reception-relevant events are

— communication element start on channel B (signal *CE start on B*, signal *idle end on B*),

— potential idle start on channel B (signal *potential idle start on B*),

— channel idle recognition point detected on channel B (signal *CHIRP on B*),

— DTS high bit received on channel B (signal *DTS received on B*, only in the dynamic segment), and

— bit strobed on channel B (signal *bit strobed on B*, not shown in Figure 93).

The channel-specific BITSTRB processes will output a *potential idle start* signal every time a bit strobed as high was preceded by a bit strobed as low. To keep the figure simple these signals are not shown in Figure 93 with the exception of the potential idle start at the end of channel activity.

#### 9.2.3.3 Transmission-related events

Figure 94 depicts the transmission-related events that are significant for media access control.

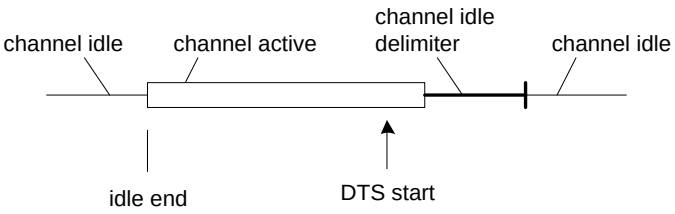



**Figure 94 — Transmission-related events for MAC**

For communication channel A the transmission-relevant events are

— the start of transmission ends the channel idle condition (signal *idle end on A*) and

— the start of the dynamic trailing sequence within the transmission pattern on channel A (signal *DTS start on A*).

For communication channel B the transmission-relevant events are

— the start of transmission ends the channel idle condition (signal *idle end on B*) and

— the start of the dynamic trailing sequence within the transmission pattern on channel B (signal *DTS start on B*).

#### 9.2.3.4 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).

## 9.3 Media access control process

### 9.3.1 States of the media access control process

This subclause 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 states

— a *MAC:standby* state,

— a *MAC:wait for CAS action point* state,

— a *MAC:wait for the cycle start* state,

— a *MAC:wait for the action point* state,

— a *MAC:wait for the static slot boundary* state,

— a *MAC:wait for the AP transmission start* state,

— a *MAC:wait for the DTS start* state,

— a *MAC:wait for the AP transmission end* state,

— a *MAC:wait for the end of the dynamic segment* state,

— a *MAC:wait for the end of the minislot* state,

— a *MAC:wait for the end of activity* state,

— a *MAC:wait for the end of the dynamic slot* state,

— a *MAC:wait for the symbol window action point* state, and

— a *MAC:wait for the end of the symbol window* state.

### 9.3.2 Initialisation and MAC:standby state

Figure 95 depicts the specification of the media access process.

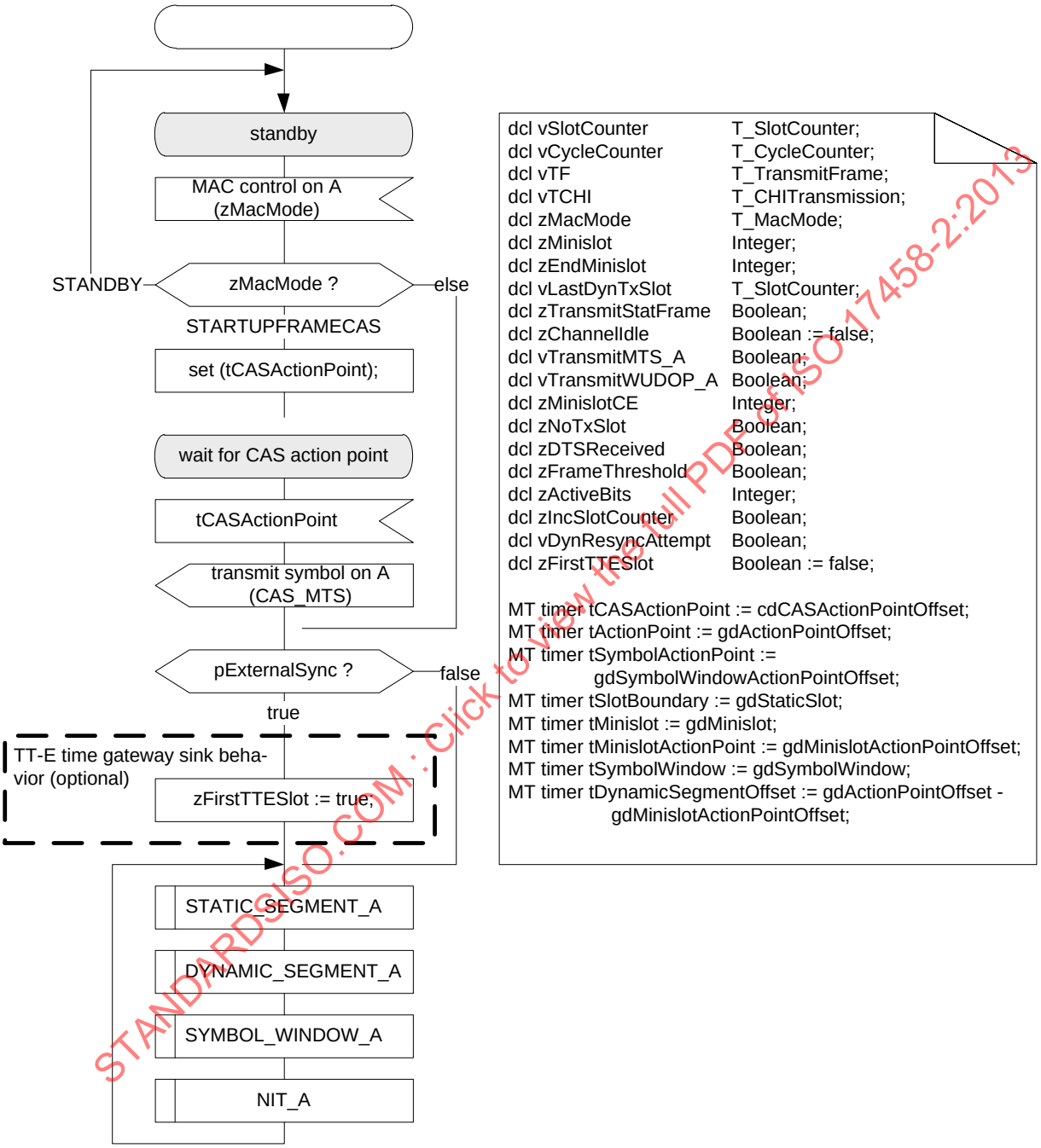





**Figure 95 — Media access process [MAC_A]**

Figure 96 depicts how mode changes are processed by the media access process.

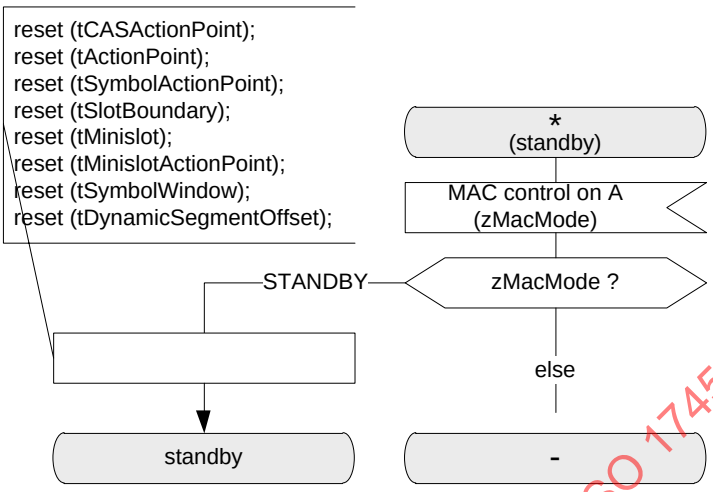



**Figure 96 — Media access control [MAC_A]**

As depicted in Figure 97 a node shall terminate the MAC process upon occurrence of the terminate event issued by the protocol operation control.

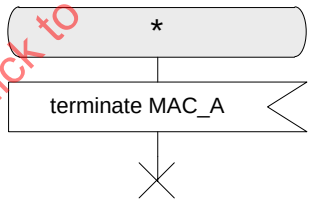



**Figure 97 — Termination of the MAC process [MAC_A]**



### 9.3.3 Static segment related states

#### 9.3.3.1 State machine for the static segment media access control

The node shall perform media access in the static segment as depicted in Figure 98.

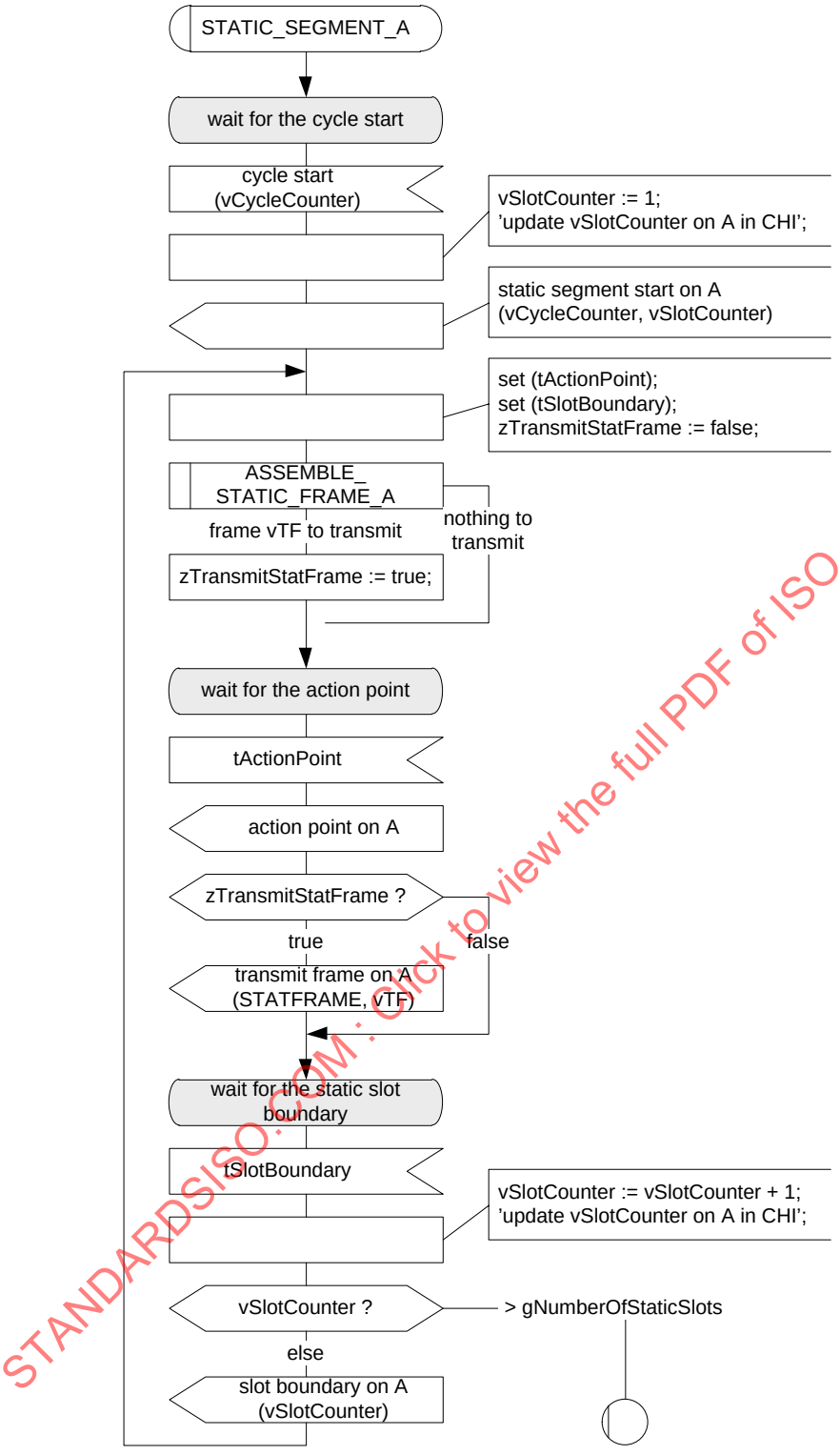



**Figure 98 — Media access in the static segment [MAC_A]**

The node shall start frame transmission at the action point of an assigned static slot if appropriate transmission conditions are met (see subclause 9.3.3.2).

The transmission data that shall be sent is specified in the *T_TransmitFrame* data structure.

**Definition: *T_SlotCounter*** **(35)**

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

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.

#### 9.3.3.2 Transmission conditions and frame assembly in the static segment

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

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

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

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

**Definition: *T_CHITransmission*** **(36)**

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

**Definition: *T_Assignment*** **(37)**

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

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.

a) The reserved bit shall be set to zero.

b) If vSlotCounter equals *pKeySlotID*, or if *pTwoKeySlotMode* is true and *vSlotCounter* equals *pSecondKeySlotID*, then

1) the startup frame indicator shall be set in accordance with *pKeySlotUsedForStartup*, and

2) the sync frame indicator shall be set in accordance with *pKeySlotUsedForSync*.

else

3) the startup frame indicator shall be set to zero, and

4) the sync frame indicator shall be set to zero.

c) The frame ID field shall be set to the current value of the slot counter *vSlotCounter*.

d) The length field shall be set to *gPayloadLengthStatic*.

e) The header CRC shall be set to the value *vTCHI!HeaderCRC* retrieved from the CHI.

f) The cycle count field shall be set to the current value of the cycle counter *vCycleCounter*.

g) If the host has data available (*vTCHI!TxMessageAvailable* set to true) then

1) the null frame indicator shall be set to one, and

2) the payload preamble indicator shall be set to the value *vTCHI!PPIndicator* imported from the CHI, and

3) if *gPayloadLengthStatic* > *vTCHI!Length* then the *vTCHI!Length* number of two-byte payload words shall be copied from *vTCHI!Message* to *vTF!Payload*.
The remaining (*gPayloadLengthStatic* - *vTCHI!Length*) two-byte payload words in *vTF!Payload* shall be set to the padding pattern 0x0000.

else if *gPayloadLengthStatic* = *vTCHI!Length* then *vTCHI!Length* number of two-byte payload words shall be copied from *vTCHI!Message* to *vTF!Payload*

else the first *gPayloadLengthStatic* number of two-byte payload words shall be copied from *vTCHI!Message* to *vTF!Payload*.

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

4) the null frame indicator shall be set to zero, and

5) the payload preamble indicator shall be set to zero, and

6) *gPayloadLengthStatic* number of two-byte payload words in *vTF!Payload* shall be set to the padding pattern 0x0000.

Figure 99 depicts the frame assembly in the static segment [MAC_A].

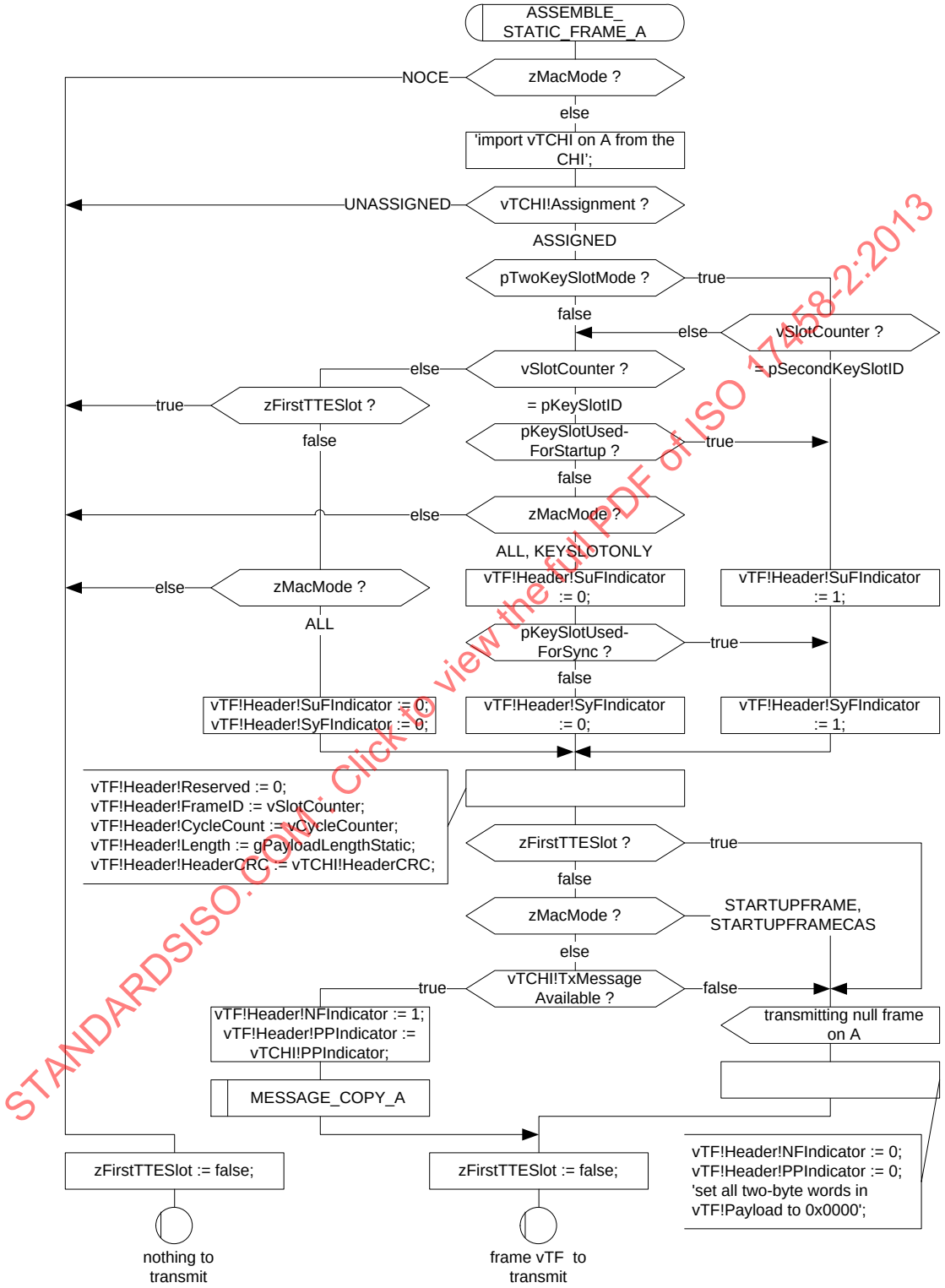



**Figure 99 — Frame assembly in the static segment [MAC_A]**

The handling of a transmission in the first static slot in the first cycle following the startup of a TT-E coldstart node is different from the handling in subsequent slots and cycles[77]. In the first slot after a TT-E coldstart node's transition from the *POC:external startup* state to the *POC:normal active* state a TT-E coldstart node will only transmit a null frame, and only if this slot is a key slot, even if the media access is operating in the ALL mode. This is controlled by the variable *zFirstTTESlot*, which is set to true before the first cycle and set to false during the first slot. This variable enables the transmission of a null frame if the first slot is a key slot and disables transmission if the first slot is not a key slot.

Figure 100 depicts the message copying and padding in the static segment [MAC_A].

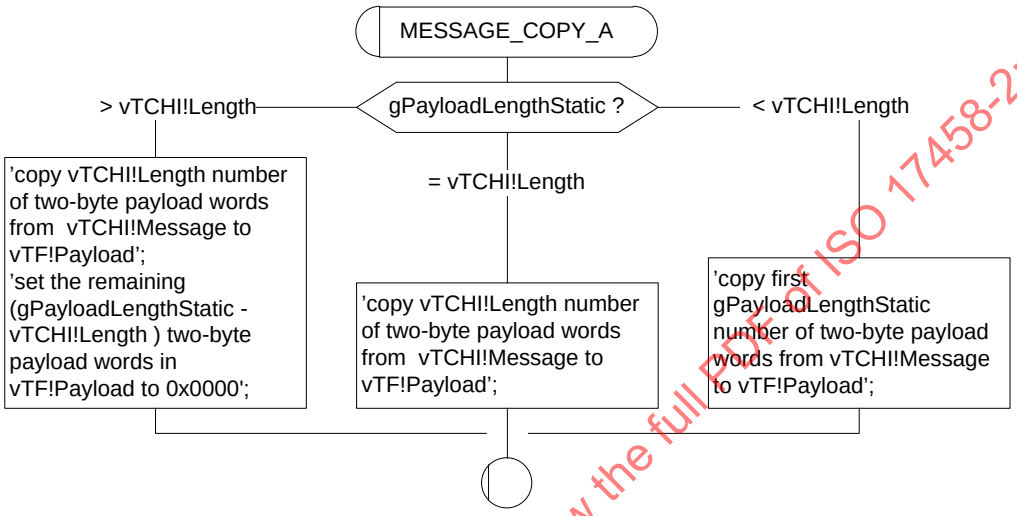



**Figure 100 — Message copying and padding in the static segment [MAC_A]**



### 9.3.4 Dynamic segment related states

#### 9.3.4.1 State machine for the dynamic segment media access control

The node shall perform media access in the dynamic segment as depicted in Figure 101 and subsequent Figures.

In the dynamic segment the node shall increment the slot counter *vSlotCounter* at the end of each dynamic slot. If the increment would cause *vSlotCounter* to exceed the maximum slot ID *cSlotIDMax* then *vSlotCounter* is instead set to zero and remains at this value until the end of the dynamic segment.

77) The special handling in the first slot of the first cycle is necessary because of the short time between the cycle start signal from the time gateway source node and the start of the cycle in the time gateway sink node. This short time would make it difficult for a practical implementation to identify and select the buffer related to the first static slot. Limiting transmissions to null frames in the first slot, if this slot is a key slot, makes this task easier, and does not have a substantial impact on cluster startup.

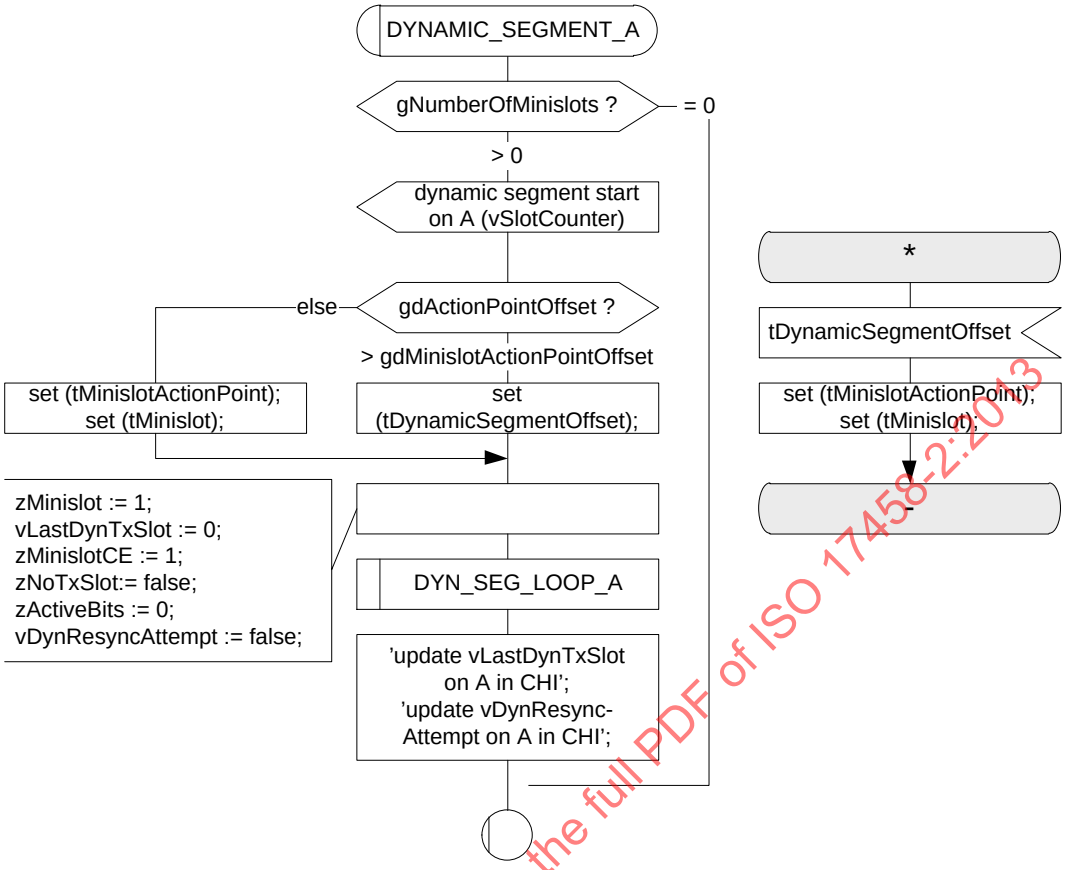



**Figure 101 — Media access in the dynamic segment [MAC_A]**

The macro DYN_SEG_LOOP keeps track of the dynamic slot counter during the dynamic segment (see Figure 103).

At the start of each dynamic slot, the node checks whether there is still enough time left in the dynamic segment for a transmission or if no transmission is allowed in this dynamic slot due to the detection of a possible slot counter desynchronisation (as indicated by the variable *zNoTxSlot*).

If a transmission is allowed, the node determines whether it has the right and need to transmit in the current dynamic slot and, if yes, does so. The transmission is described by the TRANSMIT_DYNAMIC_FRAME macro (see Figure 104).

If the node does not transmit itself, it awaits transmissions of other nodes in the MAC:wait for the end of the minislot state. Should no *CE start* signal be detected before the end of the minislot, the node proceeds to the next dynamic slot. If a *CE start* signal is detected, the node notes the current minislot and starts to count the bits of the incoming communication element.

The length of the incoming transmission is used as an indication of whether the incoming communication element is a dynamic frame or perhaps induced noise on the physical channel. Should the communication element end before the number of bits crosses the *cFrameThreshold*, the communication element is regarded as noise and the node tries to switch to a state where no noise was received.

It does so by not applying the *gdDynamicSlotIdlePhase* lengthening of the dynamic slot on the one hand and by increasing the dynamic slot counter by two should a minislot boundary have occurred between the *CE start* signal and the *CHIRP* signal. The last *potential idle start* signal before the *CHIRP* signal marks the minislot in which the frame transmission ended, and is used to derive the last minislot of the dynamic slot.

A fault-free frame reception will also enable the detection of the DTS, which is indicated by the CODEC process with the *DTS received* signal. As soon as the DTS was received, the node locks down the end of the dynamic slot, with the intent that potential noise during the succeeding idle detection cannot affect the remaining dynamic slot length.

After the reception of the *CHIRP* signal, the node awaits the end of the dynamic slot. A *CE start* signal at this point in time is generally an indication of a fault on the bus; either the preceding or the current communication element was noise or a frame transmitted due to a fault condition. In case that the preceding element was already categorized as noise due to its short length, the node treats the new communication element as frame and potentially adjusts the dynamic slot counter.

Under normal circumstances (i.e., in the noise-free case), no *CE start* signal will be received during the MAC:wait for the end of the dynamic slot state and the dynamic slot will end at the end of the minislot where *zMinislot* is equal to *zEndMinislot*. The end of the dynamic slot causes the dynamic slot counter to be incremented and then exported to the CHI.

If the received communication element was shorter than the frame threshold *cFrameThreshold* and the dynamic slot was either one or two minislots long the node will abstain from transmitting in the following dynamic slot and a resynchronisation attempt is noted for indication to the CHI at the end of the dynamic segment.

If the received communication element was shorter than the frame threshold and the dynamic slot was two minislots long the dynamic slot counter is incremented twice instead of just once, as is normally the case.

Figure 102 depicts the channel idle tracking.

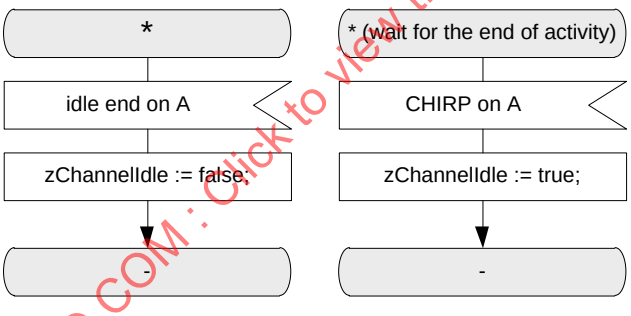



**Figure 102 — Channel idle tracking [MAC_A]**

Figure 103 depicts the media access in the dynamic segment arbitration.

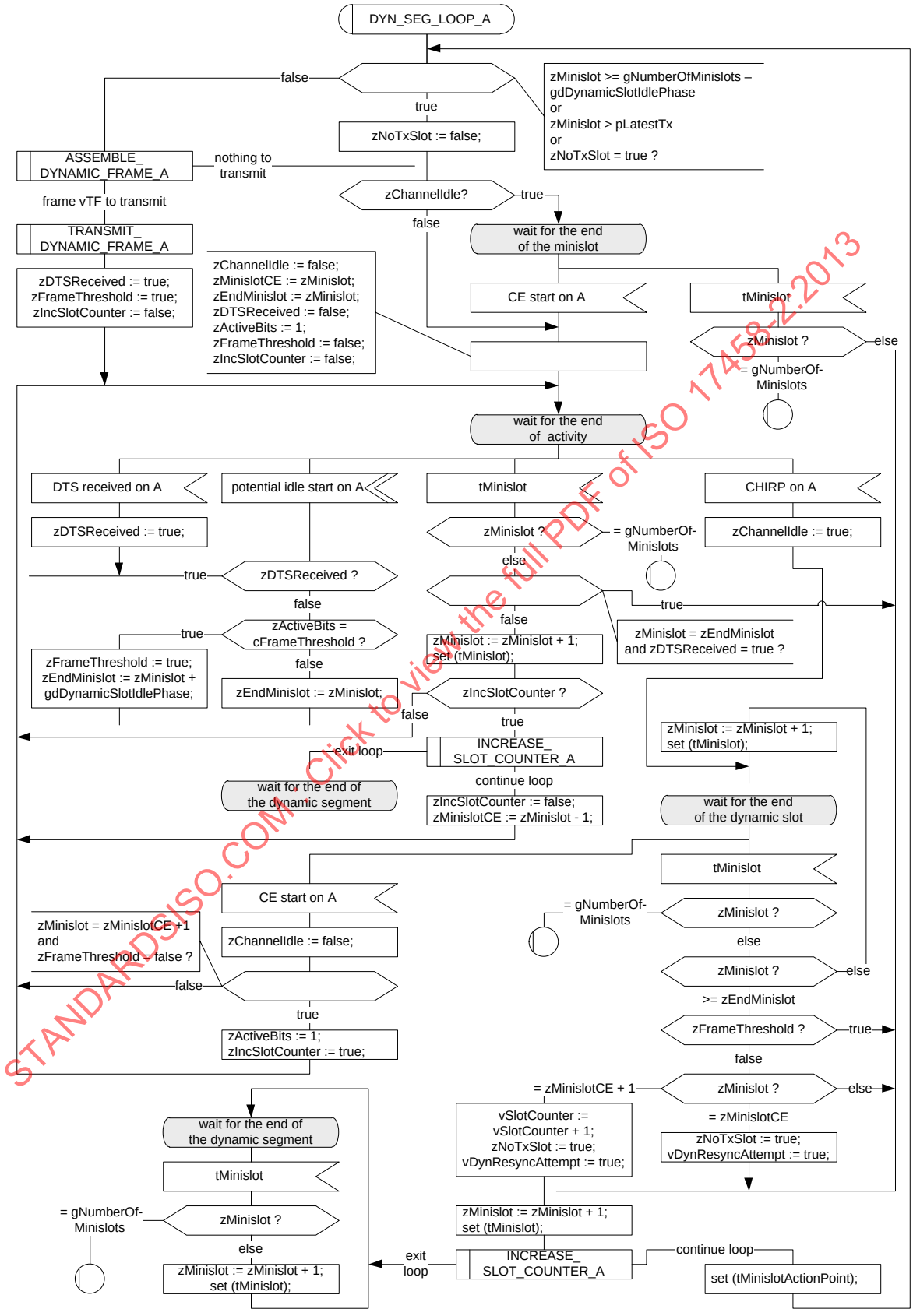



**Figure 103 — Media access in the dynamic segment arbitration [MAC_A]**

Figure 104 depicts the transmission in the dynamic segment macro.

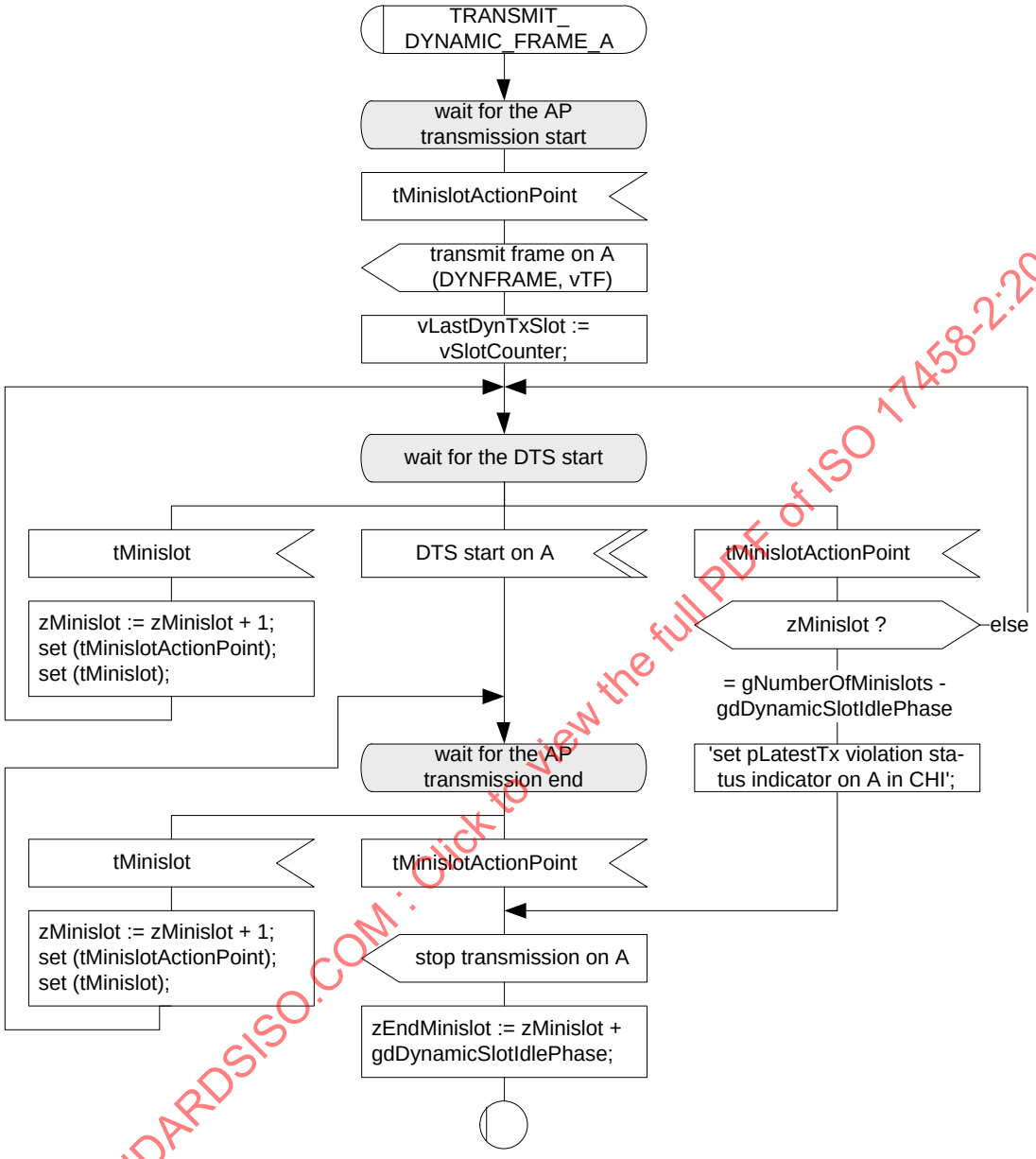



**Figure 104 — Transmission in the dynamic segment macro [MAC_A]**

Figure 105 depicts the slot counter increase macro.

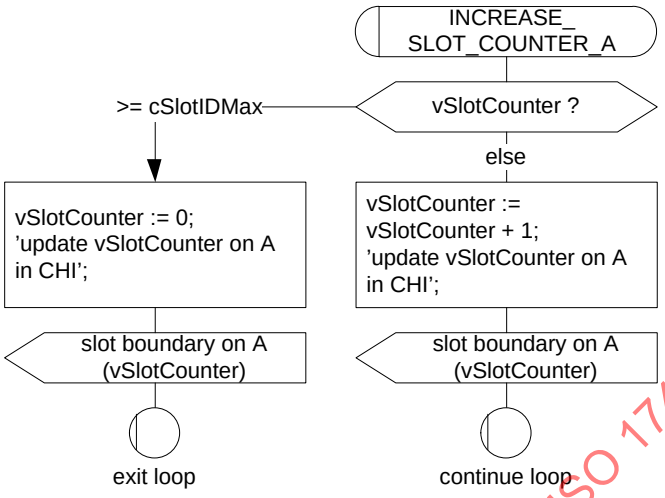



**Figure 105 — Slot counter increase macro [MAC_A]**

The INCREASE_SLOT_COUNTER_A macro is called at the end of each dynamic slot to increase the dynamic slot counter and to communicate it to the CHI. If the dynamic slot counter were to surpass the maximum slot number *cSlotIDMax*, the slot counter shall not be incremented, but is instead reset to zero.

The node shall then await the end of the dynamic segment.

Figure 106 depicts the counting of active bits.

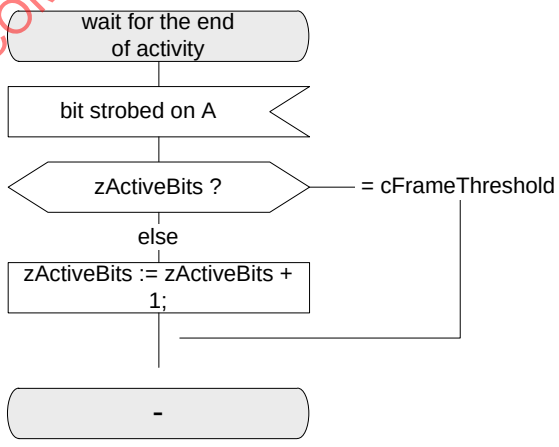



**Figure 106 — Counting of active bits [MAC_A]**

To categorize an incoming communication element as noise, the node determines whether it is shorter than the frame threshold *cFrameThreshold*.

#### 9.3.4.2 Transmission conditions and frame assembly in the dynamic segment

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

In the dynamic segment, the node shall only transmit a frame on a channel if all following conditions are fulfilled:

— the media access is operating in the ALL mode;

— the current minislot number is not larger than *pLatestTx* minislot (a node-specific upper bound);

— the slot is assigned to the node;

— consistent payload data can be imported from the CHI;

— transmission is not prohibited in the slot by the zNoTxSlot variable (i.e., zNoTxSlot was not set to true in the previous slot);

— there is at least a number of minislots equivalent to gdDynamicSlotIdlePhase before the end of the dynamic segment.

Assuming a variable *vTCHI* of type *T_CHITransmission* imported from the CHI the node shall assemble the frame in the following way if *vTCHI!Assignment* equals ASSIGNED and *vTCHI!TxMessageAvailable* equals true:

a) the reserved bit shall be set to zero;

b) the sync frame indicator shall be set to zero;

c) the startup frame indicator shall be set to zero;

d) the payload preamble indicator shall be set to the value *vTCHI!PPIndicator* retrieved from the CHI;

e) the frame ID field shall be set to the current value of the slot counter *vSlotCounter*;

f) the length field shall be set to *vTCHI!Length* retrieved from the CHI;

g) the header CRC shall be set to the value *vTCHI!HeaderCRC* retrieved from the CHI;

h) the cycle count field shall be set to the current value of the cycle counter *vCycleCounter*;

i) the null frame indicator shall be set to one;

j) *vTCHI!Length* number of two-byte payload words shall be copied from *vTCHI!Message* to *vTF!Payload*.

Figure 107 depicts the frame assembly in the dynamic segment.

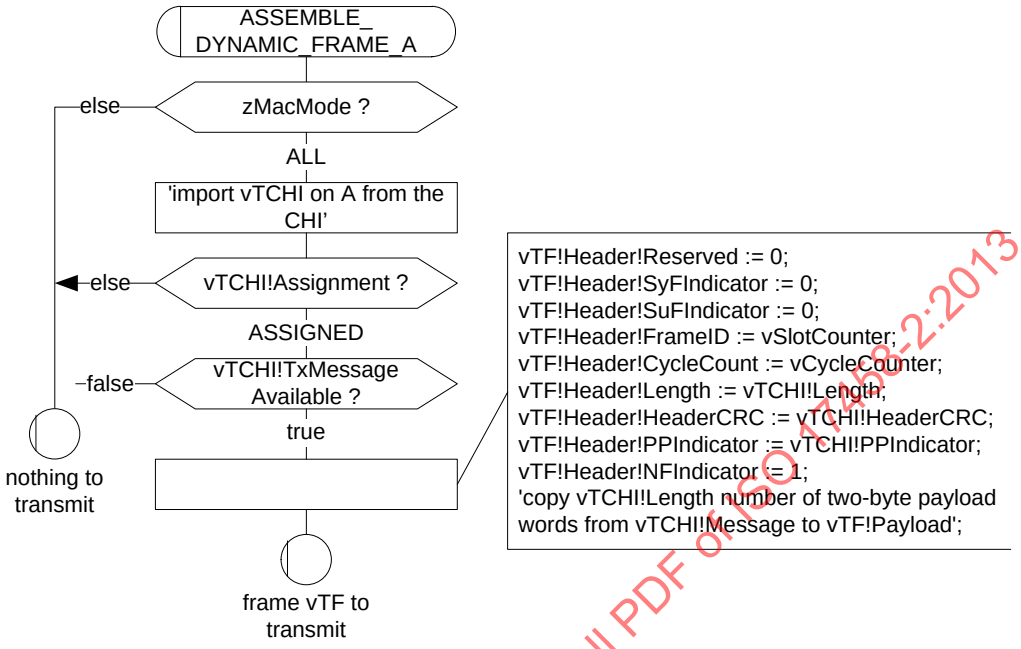



**Figure 107 — Frame assembly in the dynamic segment [MAC_A]**

### 9.3.5 Symbol window related states

The node shall perform media access in the symbol window as depicted in Figure 108.

At the start of the symbol window the node shall set the slot counter *vSlotCounter* to zero. The node shall start symbol transmission at the action point of the symbol window if the media access is in the ALL mode and if a symbol is released for transmission.

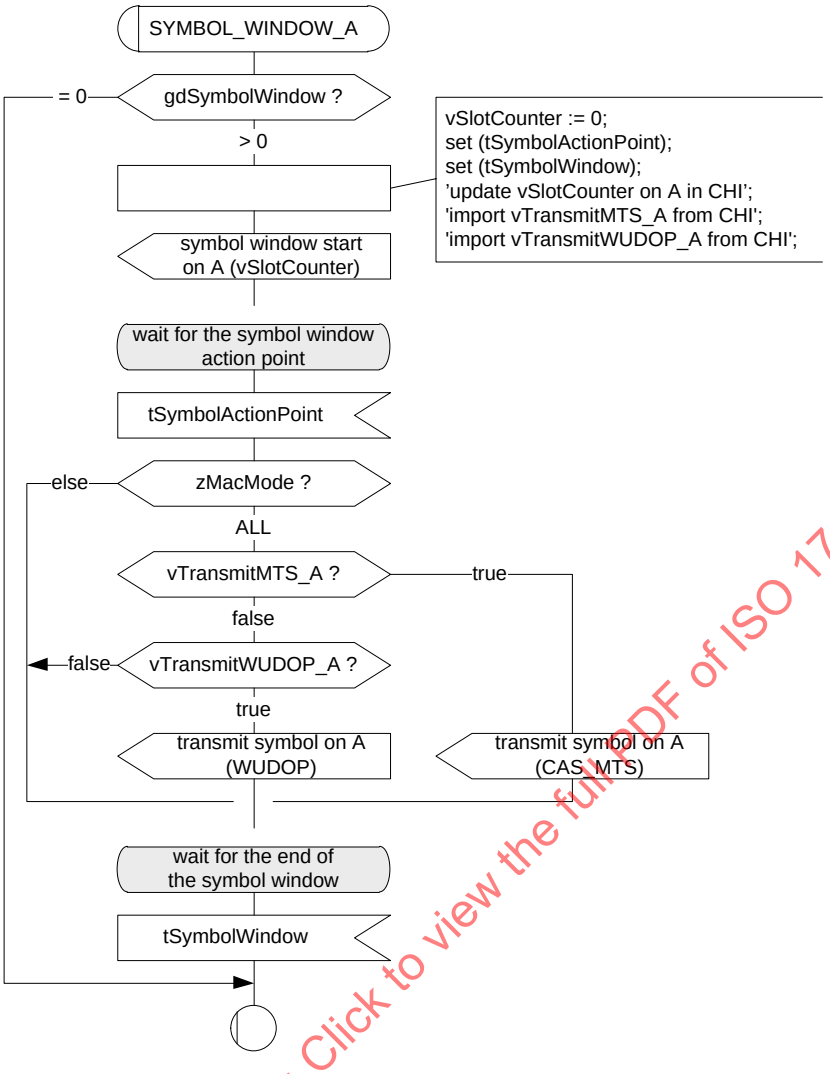



**Figure 108 — Media access in the symbol window [MAC_A]**



### 9.3.6 Network idle time

Macro NIT in Figure 109 depicts the behaviour at the start of the network idle time.

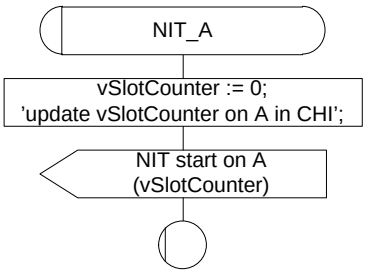



**Figure 109 — Network idle time [MAC_A]**

At the start of the NIT the node shall set the slot counter *vSlotCounter* to zero.

# 10 Frame and Symbol processing

## 10.1 Principles

Frame and symbol processing (FSP) is the main processing layer between frame and symbol decoding, which is specified in clause 7, and the controller host interface, which is specified in clause 13.

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.

## 10.2 Description

### 10.2.1 Relationship to other processes

The relationship between the Frame and Symbol Processing processes and the other protocol processes is depicted in Figure 110[78)].



78) The dark lines represent data flows between mechanisms that are relevant to this subclause. The lighter gray lines are relevant to the protocol, but not to this clause.

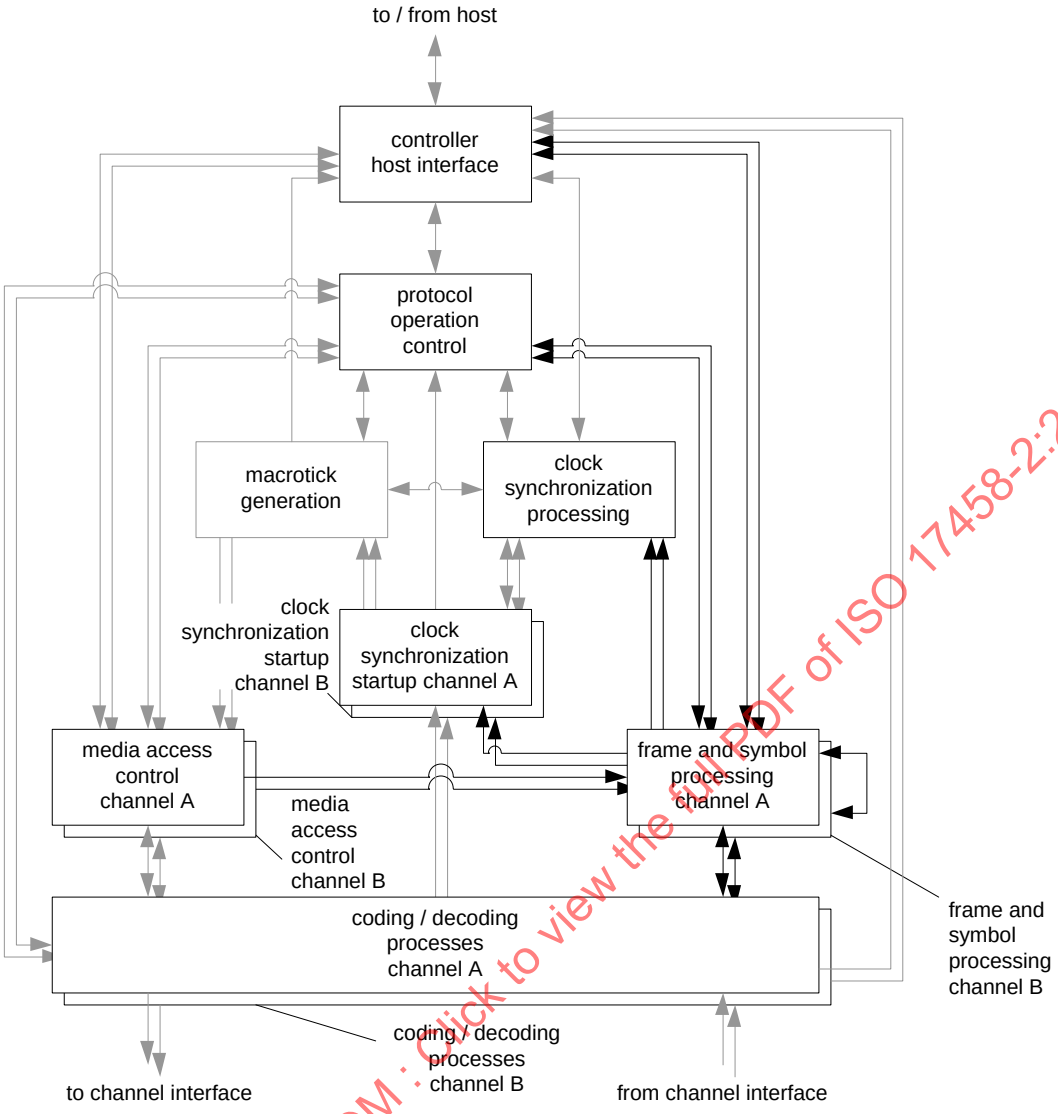



**Figure 110 — Frame and symbol processing context**



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.

### 10.2.2 Operating modes

The protocol operation control process sets the operating mode of frame and symbol processing for each communication channel.

— In the STANDBY mode the execution of the frame and symbol processing process shall be halted.

— In the STARTUP mode the frame and symbol processing process shall be executed but no update of the CHI takes place, except for decoded wakeup patterns.

— In the GO mode the frame and symbol processing process shall be executed and the update of the CHI takes place.

Definition **(38)** gives the formal definition of the FSP operating modes.

**Definition: *T_FspMode*** **(38)**

```
newtype T_FspMode
   literals STANDBY, STARTUP, GO;
endnewtype;
```

### 10.2.3 Significant events

#### 10.2.3.1 General

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.

#### 10.2.3.2 Reception-related events

Figure 111 depicts the reception-related events that are significant for frame and symbol processing.

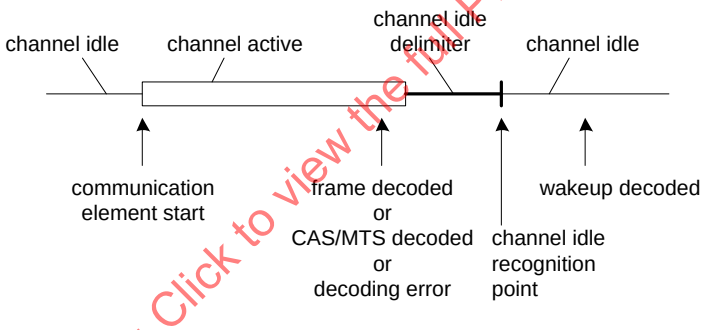



**Figure 111 — Reception-related events for FSP**

For communication channel A the reception-related events are

— communication element start on channel A (signal *CE start on A*),

— 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 **(18)**),

— CAS or MTS decoded on channel A (signal *CAS_MTS decoded on A*),

— decoding error on channel on A (signal *decoding error on A (zDecodingError)*),

— channel idle recognition point detected on channel A (signal *CHIRP on A*),

— content error on channel B (signal *content error on B*)[79], and

79) In order to address channel consistency checks for sync frames.

— wakeup decoded on channel A (signal *wakeup decoded on A*).

For communication channel B the reception-related events are

— communication element start on channel B (signal *CE start on B*),

— 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 **(18)**),

— CAS or MTS decoded on channel B (signal CAS_MTS decoded on B),

— decoding error on channel B (signal decoding error on B (*zDecodingError*)),

— channel idle recognition point detected on channel B (signal CHIRP on B),

— content error on channel A (signal content error on A), and

— wakeup decoded on channel B (signal wakeup decoded on B).

Definition **(18)** gives the formal definition of the *T_ReceiveFrame* data structure.

#### 10.2.3.3 Decoding-related events

For communication channel A the decoding-related events are

— decoding halted on channel A (signal *decoding halted on A*) and

— decoding started on channel A (signal *decoding started on A*).

For communication channel B the decoding-related events are

— decoding halted on channel B (signal *decoding halted on B*) and

— decoding started on channel B (signal *decoding started on B*).

#### 10.2.3.4 Timing-related events

Figure 112 depicts the timing-related events that are significant for frame and symbol processing.

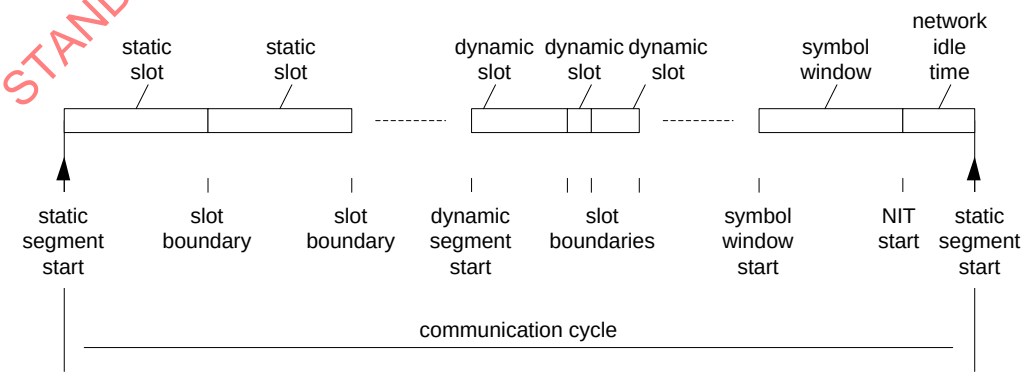



**Figure 112 — Timing-related events for FSP**

For communication channel A the relevant events are

— 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),

— 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),

— 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),

— symbol window start on channel A (signal *symbol window start on A (vSlotCounter)*; where *vSlotCounter* holds the value 0), and

— 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

— 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),

— 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),

— 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),

— symbol window start on channel B (signal *symbol window start on B (vSlotCounter)*; where *vSlotCounter* holds the value 0), and

— network idle time (NIT) start on channel B (signal *NIT start on B (vSlotCounter)*; where *vSlotCounter* holds the value 0).

### 10.2.4 Status data

For each communication channel the node shall provide a slot status that is updated in the CHI as specified in clause 13.

Definition **(39)** gives the formal definition of the slot status.

**Definition: *T_SlotStatus*** **(39)**

```
newtype T_SlotStatus
struct
    Channel          T_Channel;
    SlotCount        T_SlotCounter;
    CycleCount       T_CycleCounter;
    ValidFrame       Boolean;
    ValidMTS         Boolean;
    SyntaxError      Boolean;
    ContentError     Boolean;
    Bviolation       Boolean;
    FrameSent        Boolean;
    TxConflict       Boolean;
    NFIndicator      T_NFIndicator;
    Segment          T_Segment;
```

```
endnewtype;
```

The slot status consists of the following elements.

— *Channel* identifies the corresponding channel of the other slot status elements.

— *SlotCount* holds the value of the slot counter of the corresponding slot.

— *CycleCount* holds the value of the cycle counter of the corresponding cycle.

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

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

— *SyntaxError* denotes whether a syntax error has occurred. A syntax error occurs if any of the following criteria are met:

  — the node starts transmitting while the channel is not in the idle state;

  — a decoding error occurs;

  — a frame is decoded in the symbol window or in the network idle time;

  — a CAS / MTS is decoded in the static segment, in the dynamic segment, or in the network idle time;

  — a frame is received within the slot after the reception of a semantically correct frame;

  — an otherwise valid frame is received before transmission in a slot that is used for transmission;

  — a syntactically valid frame is detected after transmission in a slot that is used for transmission.

This element is set to false if no syntax error occurred, or to true if a syntax error did occur. Note, it is possible to have *SyntaxError* = true and *ValidFrame* = true. This could occur, for example, if a syntactically incorrect frame is received first, followed by a semantically correct and syntactically correct frame in the same slot. This would result in *ValidFrame* = true, *SyntaxError* = true, and *ContentError* = false.

— *ContentError* denotes whether a content error has occurred. A content error occurs if any of the following criteria are met:

  — in the static segment the header length contained in the header of the received frame does not match the stored header length in *gPayloadLengthStatic*;

  — 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;

  — in the static segment the null frame indicator contained in the header of the received frame is set to zero and the payload preamble indicator is set to one;

  — 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;

— 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;

— in the dynamic segment the sync frame indicator contained in the header of the received frame is set to one;

— in the dynamic segment the startup frame indicator contained in the header of the received frame is set to one;

— in the dynamic segment the null frame indicator contained in the header of the received frame is set to zero.

This element is set to false if no content error occurred, or to true if a content error did occur. It is possible to have *ContentError* = true and *ValidFrame* = true. This could occur, for example, if a syntactically correct but semantically incorrect frame is received first, followed by a semantically correct and syntactically correct frame in the same slot. This would result in *ValidFrame* = true, *SyntaxError* = false, and *ContentError* = true.

— *BViolation* denotes whether a boundary violation occurred at either beginning or end of the corresponding slot. A boundary violation occurs if the node does not consider the channel to be idle at the boundary of a slot. The element is set to false if no boundary violation occurred. See Table 9 for a description of the meaning of various combinations of *ValidFrame*, *SyntaxError*, *ContentError*, and *BViolation*.

— *FrameSent* denotes whether a non-null frame was completely transmitted in the corresponding slot. The element is set to true if there was a complete non-null frame transmission in the slot, and otherwise it is set to false.

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

— *NFIndicator* denotes whether a null frame was received. The element is set to 0 if either a null frame, an invalid frame or nothing was received, or to 1 if a valid non-null frame was received[80)].

— *Segment* denotes the segment in which the slot status was recorded.

**Definition: *T_Segment*** **(40)**

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

The element *Segment* is set to STUP during the non-synchronized startup phase. The values STATIC, DYNAMIC, SW, and NIT denote the static segment, the dynamic segment, the symbol window, and the network idle time, respectively.

## 10.3 Frame and symbol processing process

### 10.3.1 States of the frame and symbol processing process

This subclause 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 113 gives an overview of the frame and symbol processing-related state diagram.

80) The parameter *NFIndicator* is always set to 1 for valid frames in the dynamic segment.

For each communication channel the FSP process contains the five states

— *FSP:standby* state,

— *FSP:wait* for CE start state,

— *FSP:decoding* in progress state,

— *FSP:wait* for CHIRP state, and

— *FSP:wait* for transmission end state.

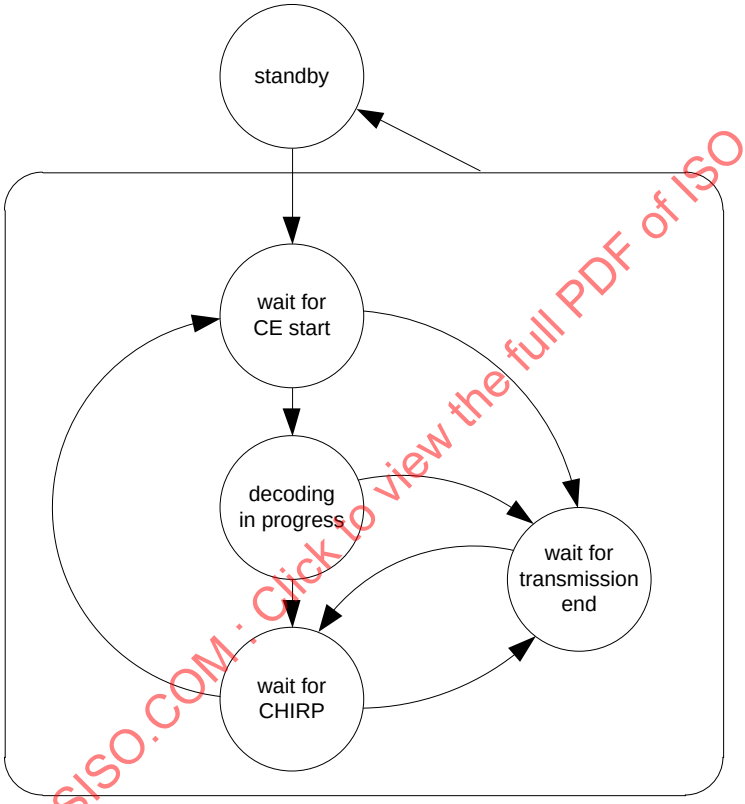



**Figure 113 — State overview of the FSP state machine (shown for one channel)**



### 10.3.2 Initialisation and FSP:standby state

As depicted in Figure 114, 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.

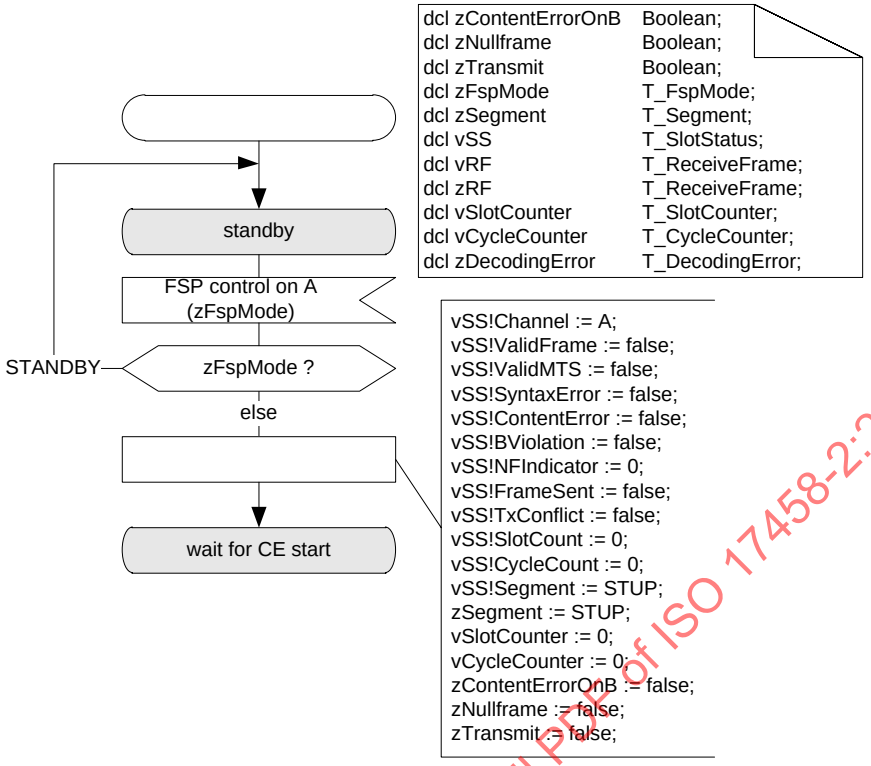



**Figure 114 — FSP process [FSP_A]**

As depicted in Figure 115, 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.

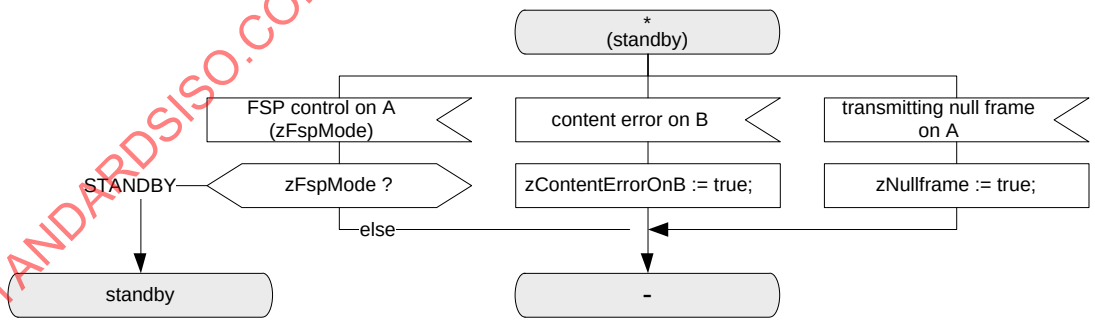



**Figure 115 — FSP control [FSP_A]**

In addition, the node shall apply cross-channel content checks to identify cross-channel inconsistencies whenever *zSegment* = STATIC[81)].

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

As depicted in Figure 116, a node shall terminate the FSP process upon occurrence of the terminate event issued by the protocol operation control process.

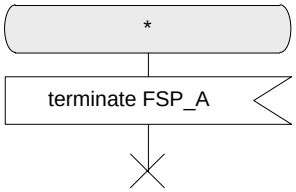



**Figure 116 — Termination of the FSP process [FSP_A]**

Depending on the operation mode of the FSP process and the segment, the FSP process relays decoded wakeup patterns from the WUPDEC process to the CHI as shown in Figure 117.

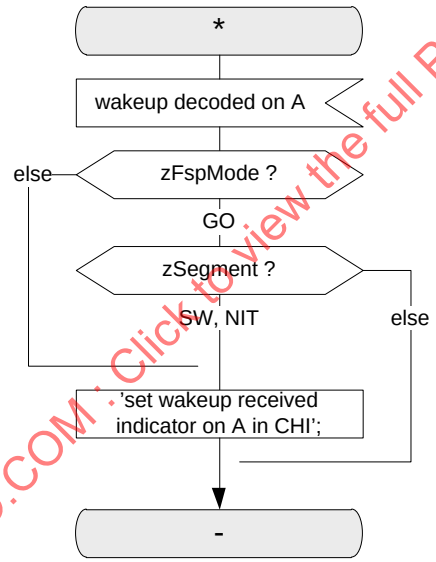



**Figure 117 — CHI update of a decoded wakeup pattern [FSP_A]**



### 10.3.3 Macro SLOT_SEGMENT_END

The macro SLOT_SEGMENT_END that is depicted in Figure 118 shall be called within the FSP process

— at the end of each static slot,

— at the end of each dynamic slot if a dynamic segment is configured (i.e., *gNumberOfMinislots* > 0),

— at the end of the symbol window if the symbol window is configured (i.e., *gdSymbolWindow* > 0), and

— 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.

Figure 118 depicts the slot and segment end macro [FSP_A].

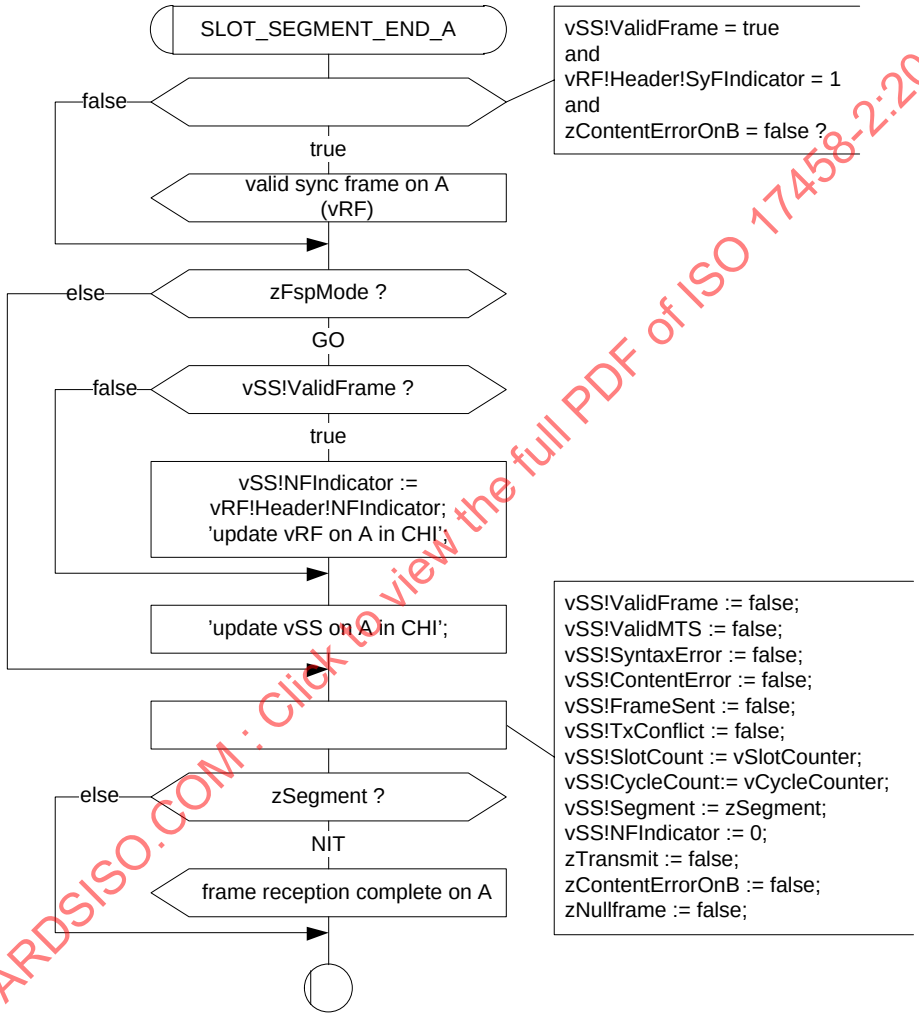





**Figure 118 — Slot and segment end macro [FSP_A]**

### 10.3.4 FSP:wait for CE start state

The *FSP:wait for CE start* state and the transitions out of this state are depicted in Figure 119.

For each configured communication channel a node shall remain in the *FSP:wait for CE start* state until either

— a communication element start is received, or

— the node starts transmitting a communication element on the channel.

If either a slot boundary or one of the four segment boundaries is crossed then the node shall execute the SLOT_SEGMENT_END macro to provide the current slot status, and any frame data that may have been received, to the host interface for further processing. In this case the node shall remain in the *FSP:wait for CE start* state.

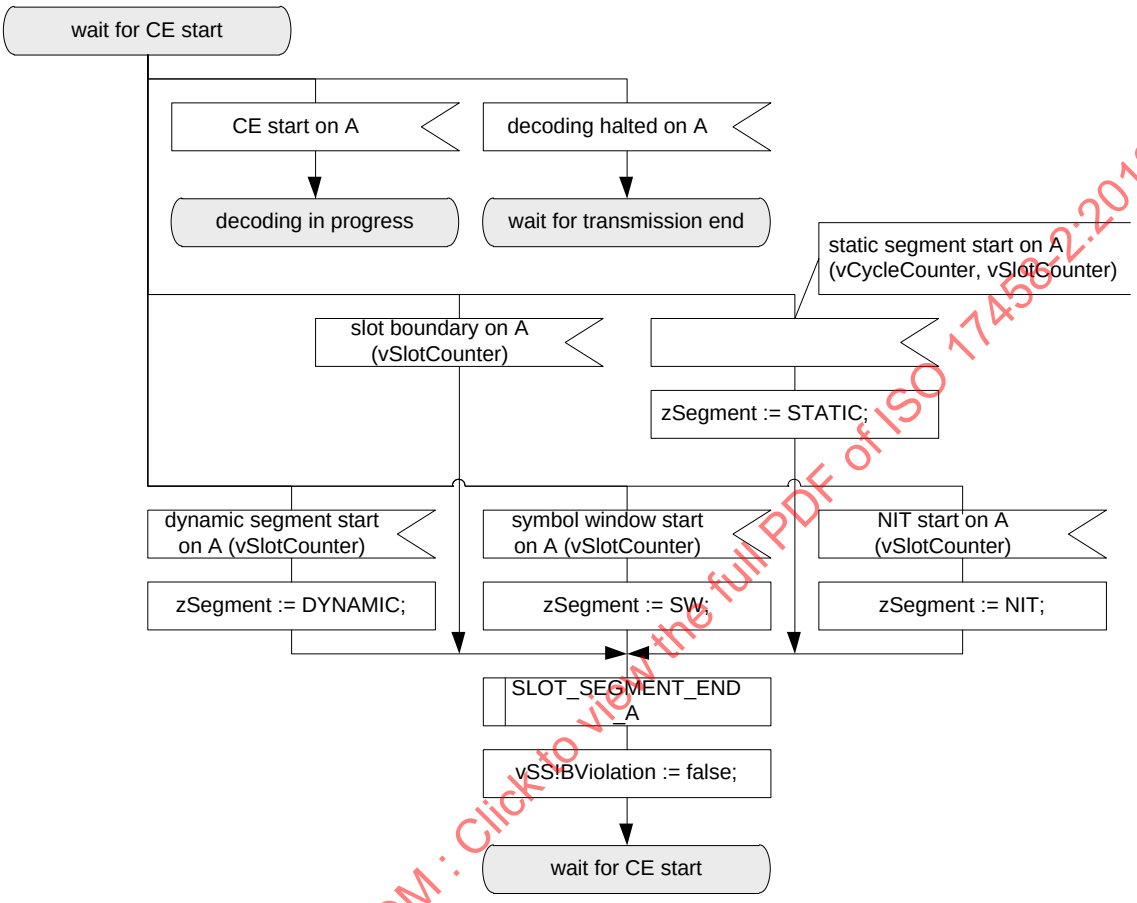



**Figure 119 — Transitions from the FSP:wait for CE start state [FSP_A]**



### 10.3.5 FSP:decoding in progress state

#### 10.3.5.1 Conditions to leave the FSP:decoding in progress state

The *FSP:decoding in progress* state and the transitions out of this state are depicted in Figure 120 and Figure 121.

For each configured communication channel a node shall remain in the *FSP:decoding* in progress state until either

— the node starts transmitting on the communication channel, or

— a decoding error occurs on the communication channel, or

— a syntactically correct frame is decoded on the communication channel, or

— a CAS / MTS symbol was decoded, or

— a slot boundary or one of the four segment boundaries is crossed.

If either a slot boundary or one of the four segment boundaries is crossed then the node shall execute the SLOT_SEGMENT_END macro to provide the current slot status, and any frame data that may have been received, to the host interface for further processing.

Figure 120 and Figure 121 are depicting the transitions from the *FSP:decoding* in progress state [FSP_A].

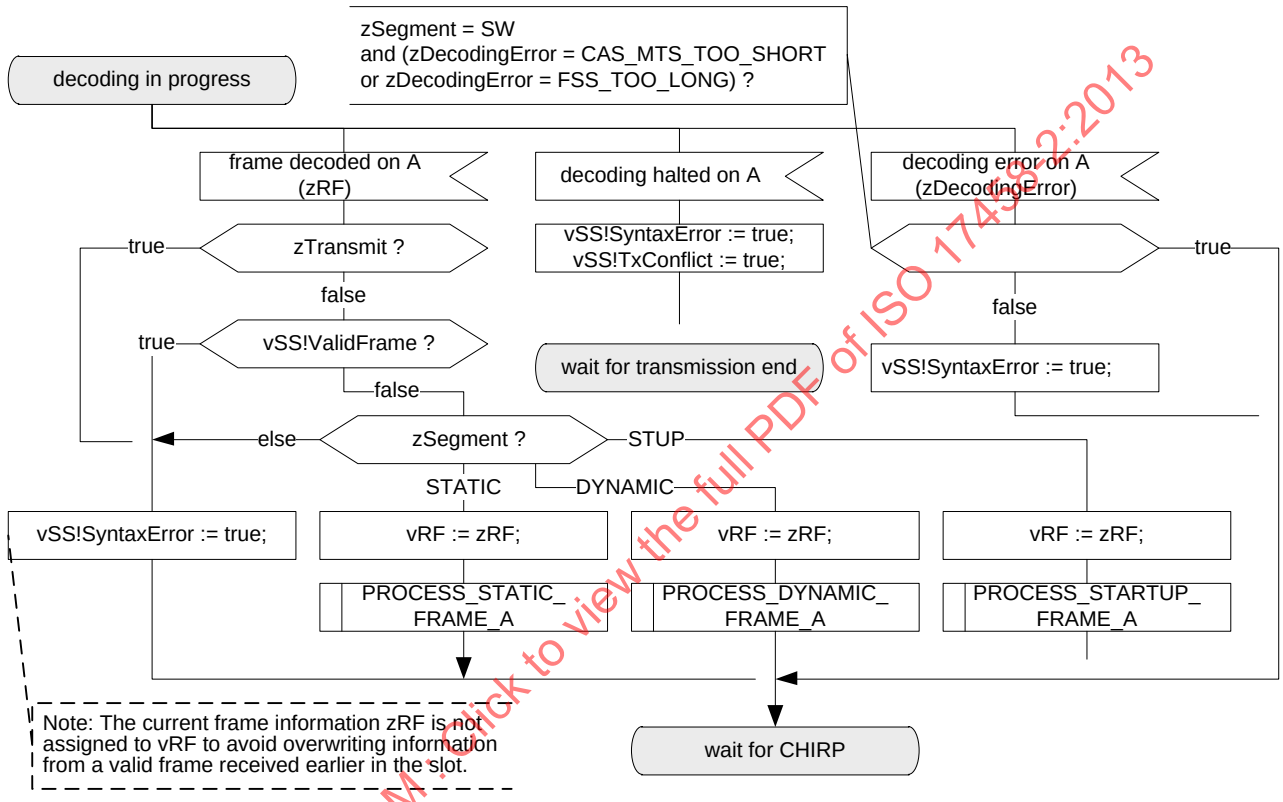



**Figure 120 — Transitions from the FSP:decoding in progress state [FSP_A]**

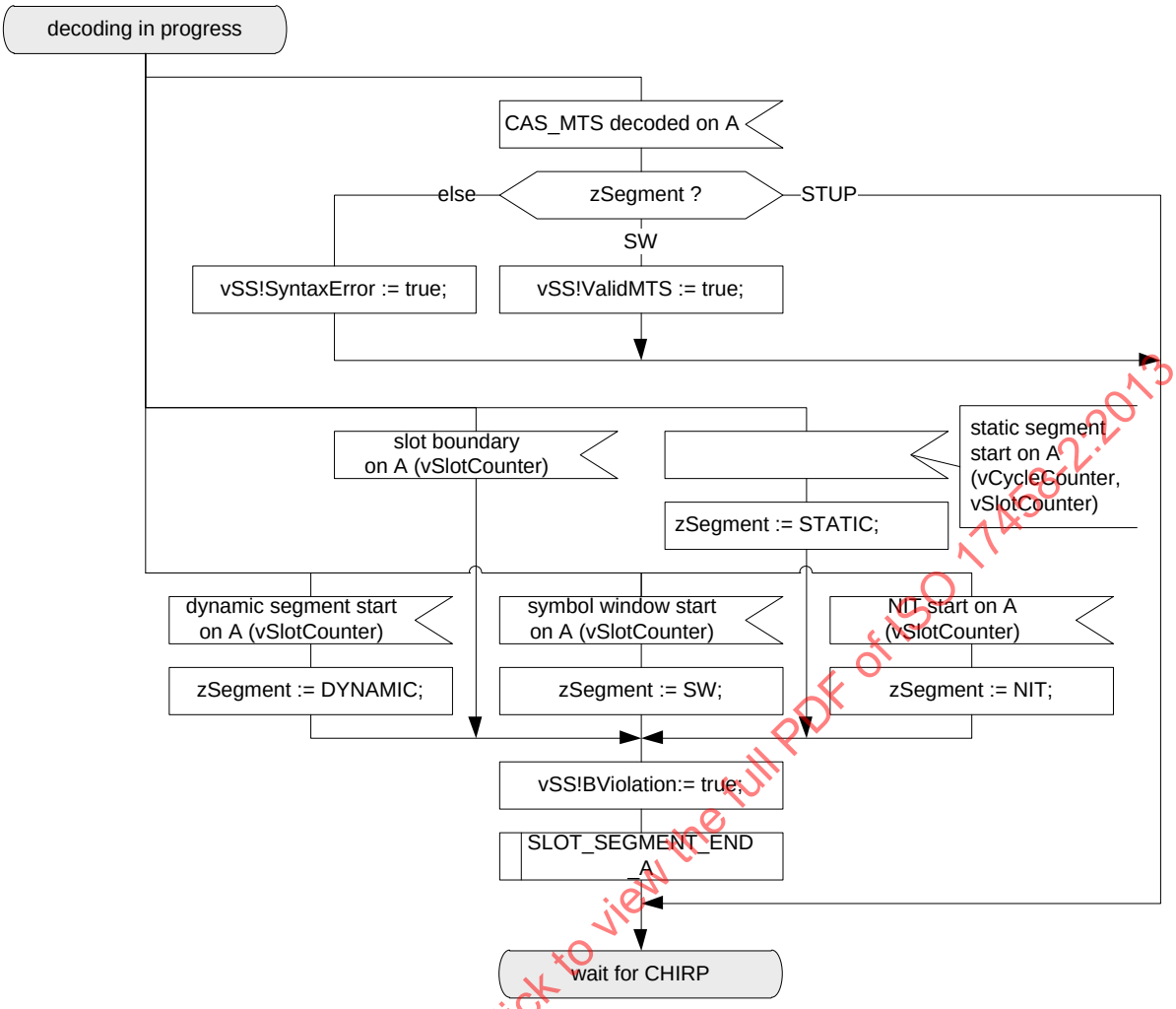



**Figure 121 — Transitions from the FSP:decoding in progress state [FSP_A]**



### 10.3.5.2 Frame reception checks during non-synchronized operation

The frame acceptance checks that the node shall apply during non-synchronized operation are defined in the macro PROCESS_STARTUP_FRAME depicted in Figure 122.

For each configured communication channel the node shall accept each frame that fulfils all of the following criteria.

— 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*.

— The payload length included in the header of the frame equals the globally configured length for static frames *gPayloadLengthStatic*.

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

— The startup frame indicator included in the header is set to one.

— If the null frame indicator *vRF!Header!NFIndicator* is set to zero then the *vRF!Header!PPIndicator* is not set to one.

— The cycle counter included in the header of the frame is not larger than the configured maximum cycle counter *gCycleCountMax*.

A frame that passes these checks is called a valid startup frame.

If the cycle count value included in the header of a valid startup frame is even then the frame is called a valid even startup frame.

If the cycle count value included in the header of a valid startup frame is odd then the frame is called a valid odd startup frame.

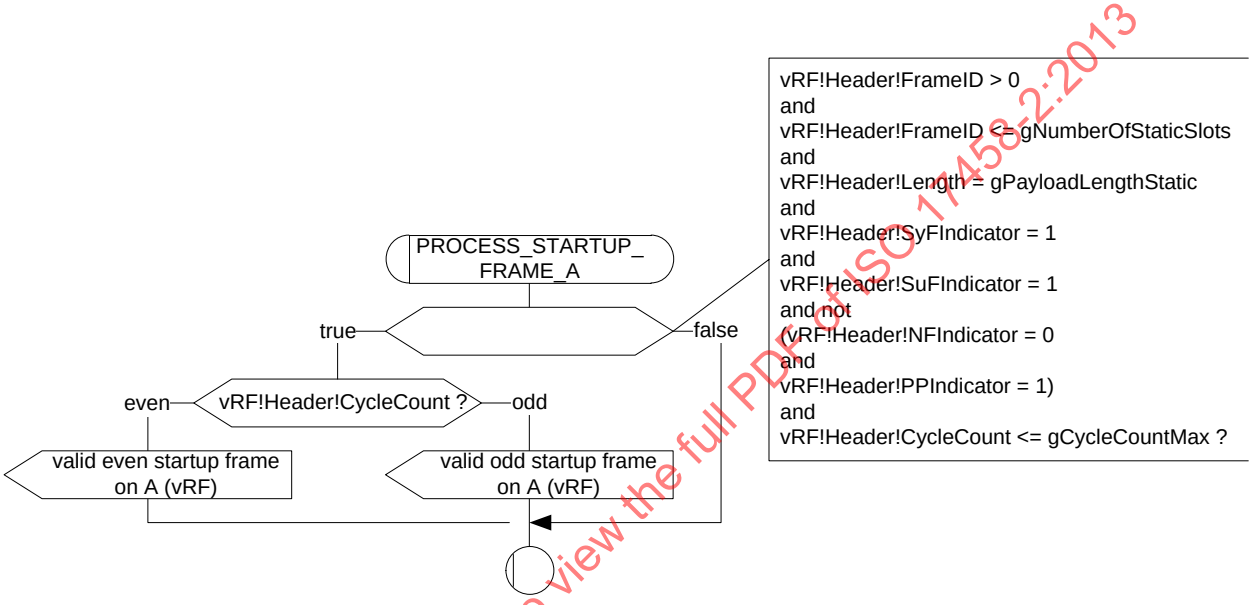



**Figure 122 — Frame acceptance checks during non-synchronized operation [FSP_A]**



#### 10.3.5.3 Frame reception checks during synchronized operation

##### 10.3.5.3.1 Frame reception checks in the static segment

Figure 123 depicts the frame reception timing that shall be met by a syntactically valid frame in the static segment.

The frame acceptance checks that the node shall apply during synchronized operation in the static segment are defined in the macro PROCESS_STATIC_FRAME depicted in Figure 124.

For each configured communication channel the node shall accept the first frame that fulfils all of the following criteria.

— The frame is contained within one static slot.

— 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*.

— The frame ID included in the header of the frame equals the value of the slot counter *vSlotCounter*.

— The cycle count included in the header of the frame matches the value of the cycle counter *vCycleCounter*.

— If the startup frame indicator in the header *vRF!Header!SuFIndicator* is set to one then the *vRF!Header!SyFIndicator* is not set to zero.

— If the null frame indicator *vRF!Header!NFIndicator* is set to zero then the *vRF!Header!PPIndicator* is not set to one.

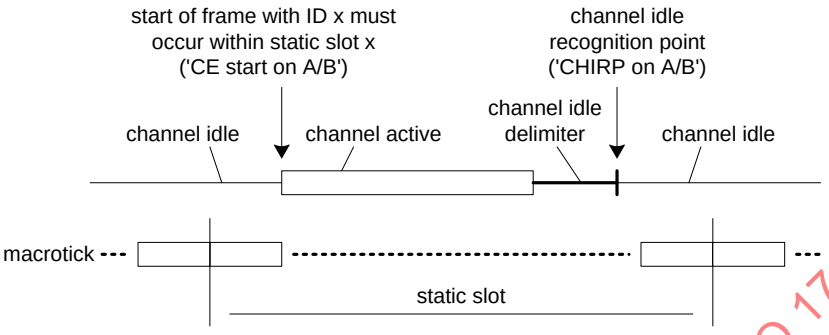



**Figure 123 — Frame reception timing for a static slot**

Figure 124 depicts the frame acceptance checks for the static segment [FSP_A].

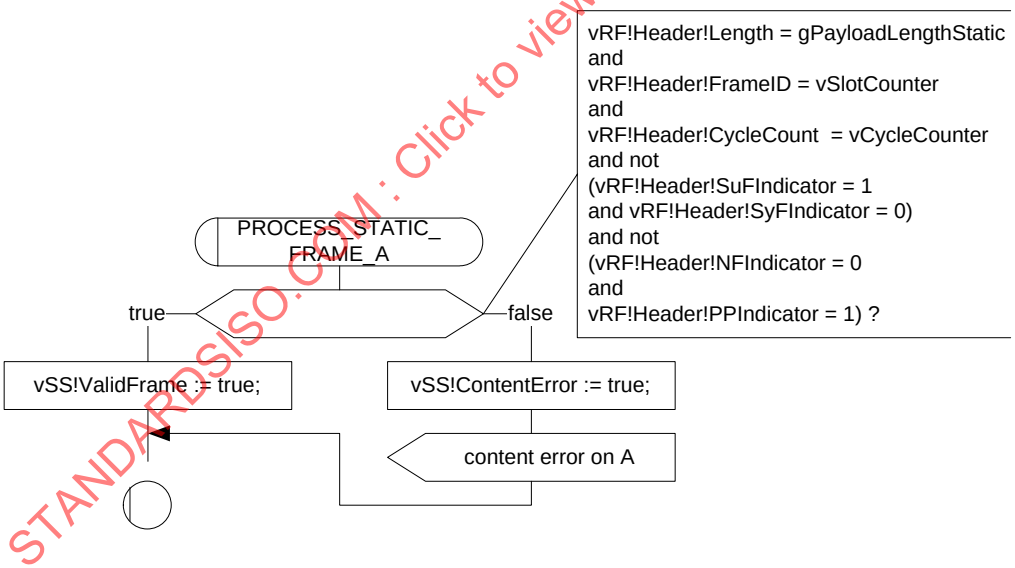



**Figure 124 — Frame acceptance checks for the static segment [FSP_A]**



#### 10.3.5.3.2 Frame reception checks in the dynamic segment

Figure 125 depicts the frame reception timing that shall be met by a syntactically valid frame in the dynamic segment.

The frame acceptance checks that the node shall apply during synchronized operation in the dynamic segment are defined in the macro PROCESS_DYNAMIC_FRAME depicted in Figure 126.

For each configured communication channel the node shall accept the first frame that fulfils all of the following criteria.

— The frame ID included in the header of the frame is greater than 0 and matches the value of the slot counter *vSlotCounter*.

— The cycle count included in the header of the frame matches the value of the cycle counter *vCycleCounter*.

— The sync frame indicator included in the header is set to zero.

— The startup frame indicator included in the header is set to zero.

— The null frame indicator included in the header is set to one.

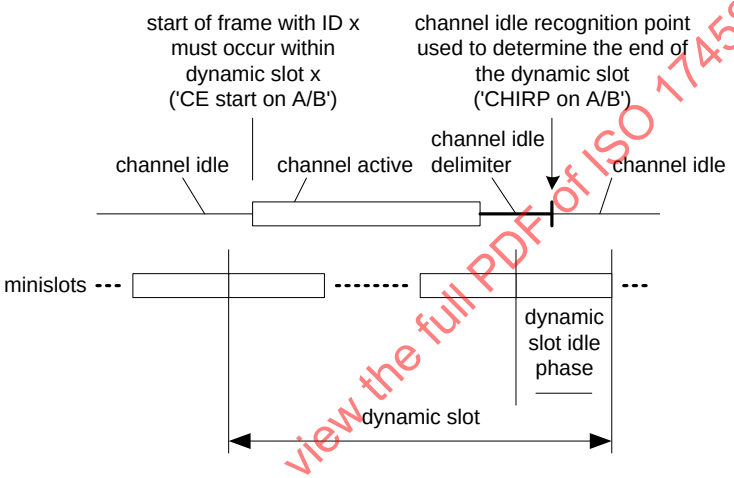



**Figure 125 — Frame reception timing for a dynamic slot**

Figure 126 depicts the frame acceptance checks for the dynamic segment [FSP_A].

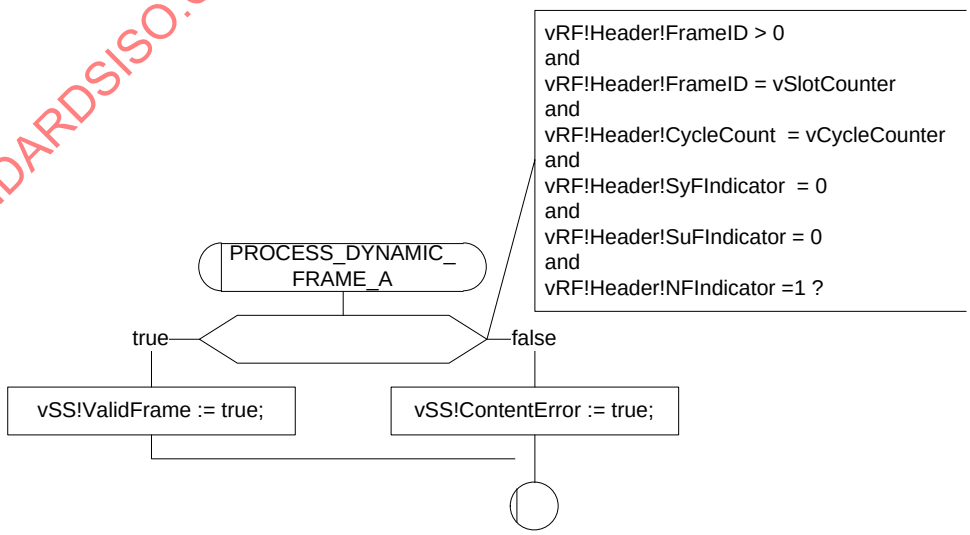



**Figure 126 — Frame acceptance checks for the dynamic segment [FSP_A]**

### 10.3.6 FSP:wait for CHIRP state

The *FSP:wait for CHIRP* state and the transitions out of this state are depicted in Figure 127.

For each configured communication channel a node shall remain in the *FSP:wait for CHIRP* state until either

— the channel idle recognition point is identified on the communication channel, or

— the node starts transmitting on the communication channel.

If either a slot boundary or one of the four segment boundaries is crossed then the node shall execute the SLOT_SEGMENT_END macro to provide the current slot status, and any frame data that may have been received, to the host interface for further processing. In this case the node shall remain in the *FSP:wait for CHIRP* state.

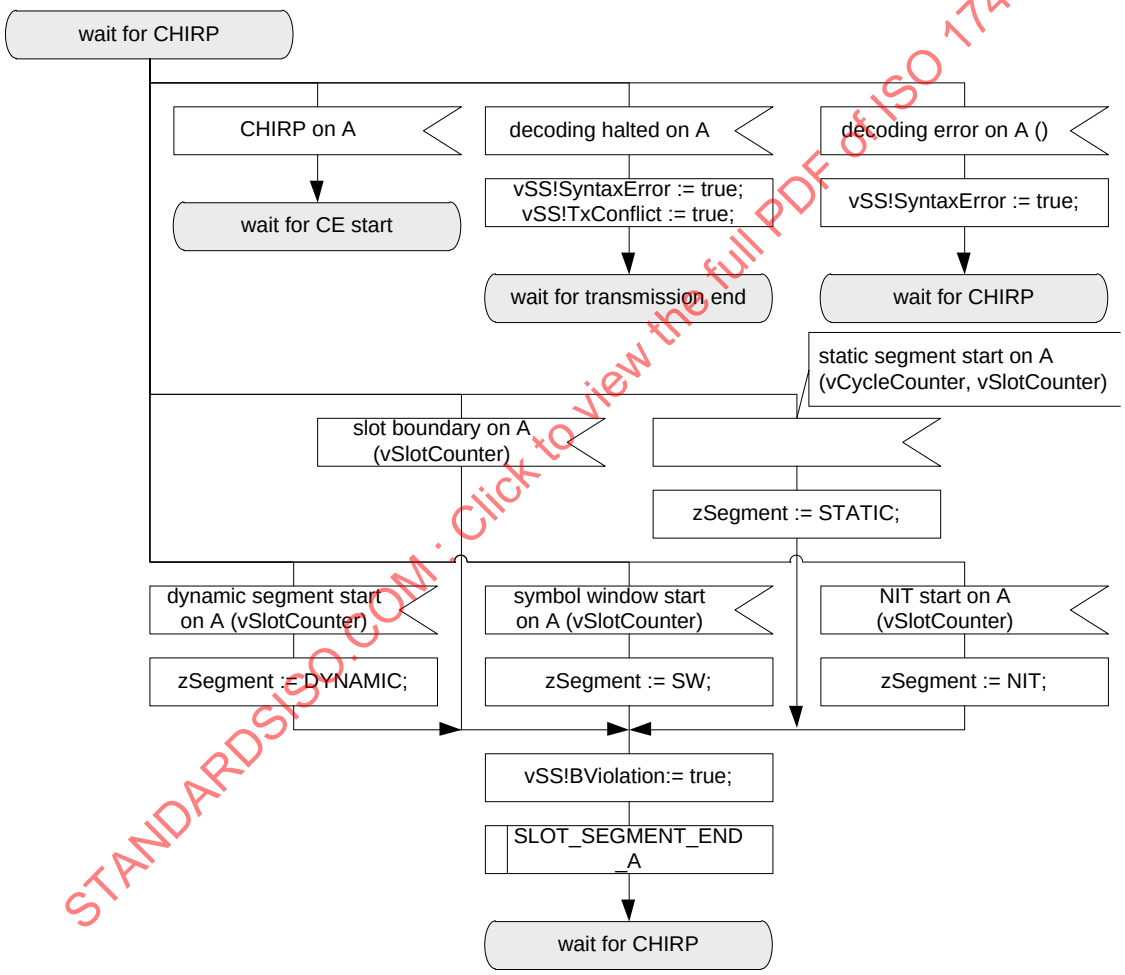



**Figure 127 — Transitions from the FSP:wait for CHIRP state [FSP_A]**

### 10.3.7 FSP:wait for transmission end state

The *FSP:wait for transmission end* state and the transitions out of this state are depicted in Figure 128.

For each configured communication channel a node shall remain in the *FSP:wait for transmission end* state until either

— the transmission ends on the channel, or

— the slot boundary or one of the four segment boundaries is crossed.

If either a slot boundary or one of the four segment boundaries is crossed then the node shall signal a fatal protocol error to the protocol operation control process.

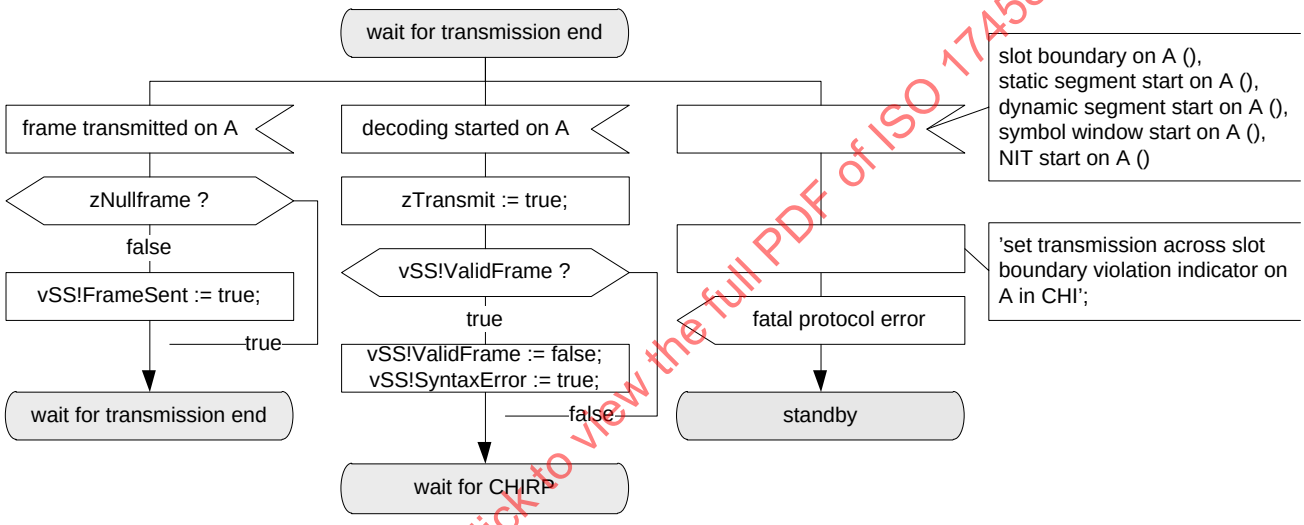



**Figure 128 — Transitions from the FSP:wait for transmission end state [FSP_A]**



# 11 Wakeup and Startup

## 11.1 General

This clause describes the protocol mechanisms available to allow a node to cause a transition of a FlexRay cluster from a sleep mode to a mode where nodes are ready to begin synchronized operation (wakeup) and to allow a node to either initiate synchronized operation or integrate into a cluster that is already operating (startup).

First the protocol mechanisms of remote cluster wakeup are described in detail. Additional material in the form of application notes related to the interaction between the communication controller and host, as well as describing techniques for wakeup during operation, may be found in Annex C.

Following the cluster wakeup subclause, communication startup is described. This subclause also describes the integration (or reintegration) of nodes into a communication cluster.

## 11.2 Cluster wakeup

### 11.2.1 Principles

This subclause describes the procedure[82)] used by communication controllers to initiate the remote cluster wakeup. The following procedures assume that the bus drivers in the system support the optional remote wakeup detection capability.

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[83)]. The communication controller provides the host the ability to transmit a special wakeup pattern (see clause 7) on each of its available channels separately.

The wakeup pattern shall 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 shall 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[84)] 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 Annex C).

The wakeup procedure tolerates any number of nodes simultaneously trying to wake up a channel and resolves this situation such that eventually only one node transmits the wakeup pattern. Additionally, the wakeup pattern is collision resilient; so even in the presence of a fault causing two nodes to simultaneously transmit a wakeup pattern the signal resulting from the collision can still wake up the other nodes.

### 11.2.2 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 129[85)].



82) 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.

83) 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.

84) For example, the transmission unit of the bus driver may be faulty.

85) The dark lines represent data flows between mechanisms that are relevant to this subclause. The lighter gray lines are relevant to the protocol, but not to this clause.

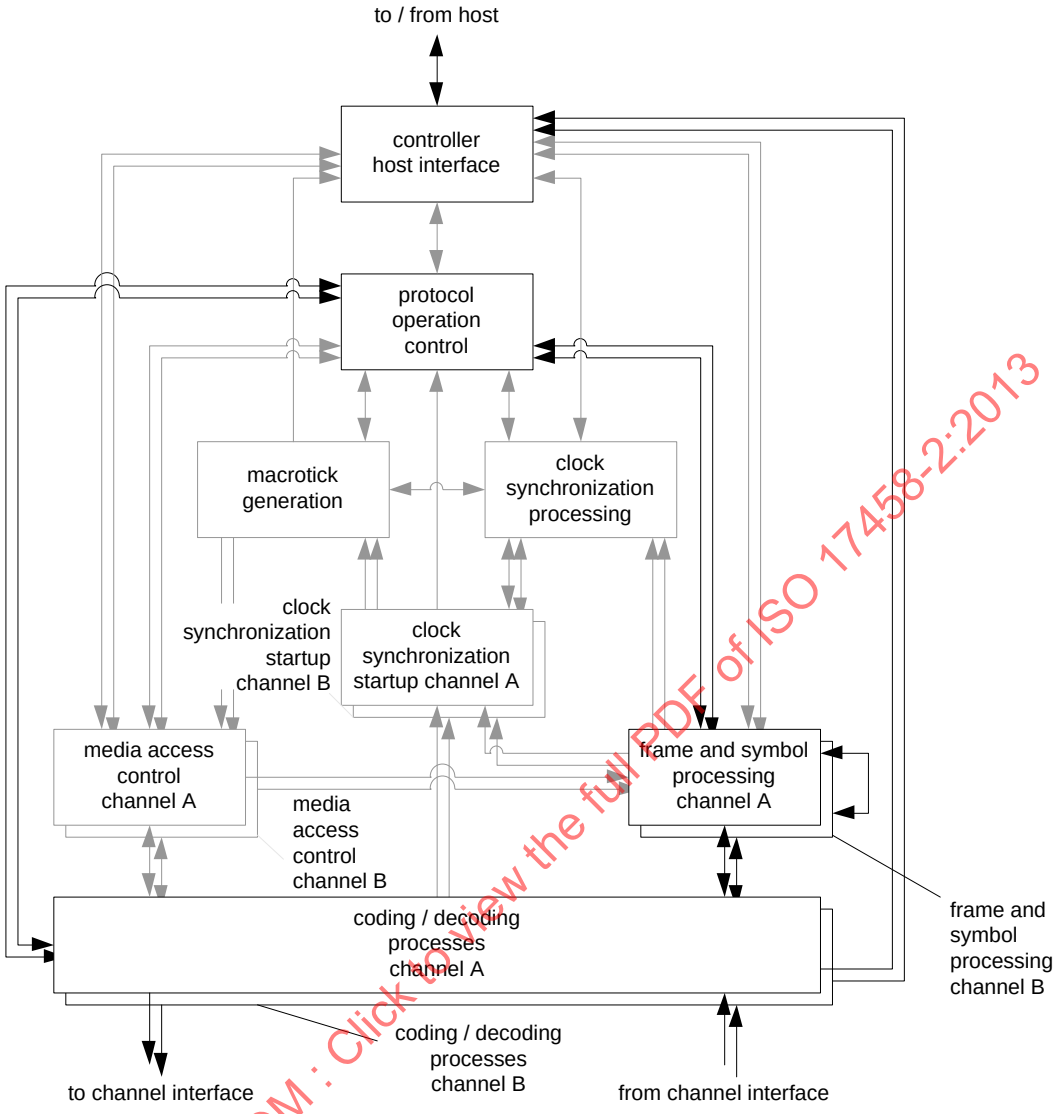



**Figure 129 — Protocol operation control context**



### 11.2.3 Wakeup support by the communication controller

#### 11.2.3.1 Host interaction

The host shall 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 130) 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 13.3.1.3.2).

The host shall 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 clause 6).

### 11.2.3.2 Wakeup state diagram

Figure 130 depicts the structure of the wakeup state machine [POC].

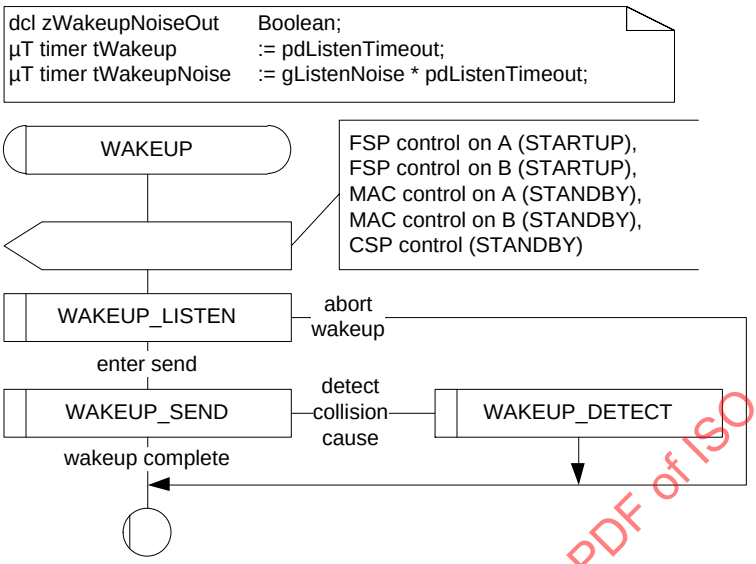



**Figure 130 — Structure of the wakeup state machine [POC]**

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 6.2.2.4 in Definition **(5)**.

The return status variable *vPOC!WakeupStatus* is set by the POC to

— UNDEFINED, if the host has not issued a WAKEUP command since the last entry to the *POC:default config* state (see Figure 29), or when the POC has not completed[86)] a wakeup requested by the host, or

— RECEIVED_HEADER, if the communication controller has received a frame header without coding violation on either channel during the initial listen phase, or

— RECEIVED_WUP, if the communication controller has received a valid wakeup pattern on channel *pWakeupChannel* during the initial listen phase, or

— COLLISION_HEADER, if the communication controller has detected a collision during wakeup pattern transmission by receiving a valid header during the ensuing detection phase, or

86) This could occur if the wakeup is still in progress when the variable is examined by the host, if the POC aborted the wakeup prior to completion due to an IMMEDIATE_READY, DEFERRED_READY, DEFFERED_HALT, or FREEZE command from the host, or if the POC did not attempt a wakeup because *pWakeupPattern* < 2.

— 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, or

— 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, or

— TRANSMITTED, if the wakeup pattern was completely transmitted.

#### 11.2.3.3 The POC:wakeup listen state

Figure 131 depicts the transitions from the *POC:wakeup listen* state [POC].

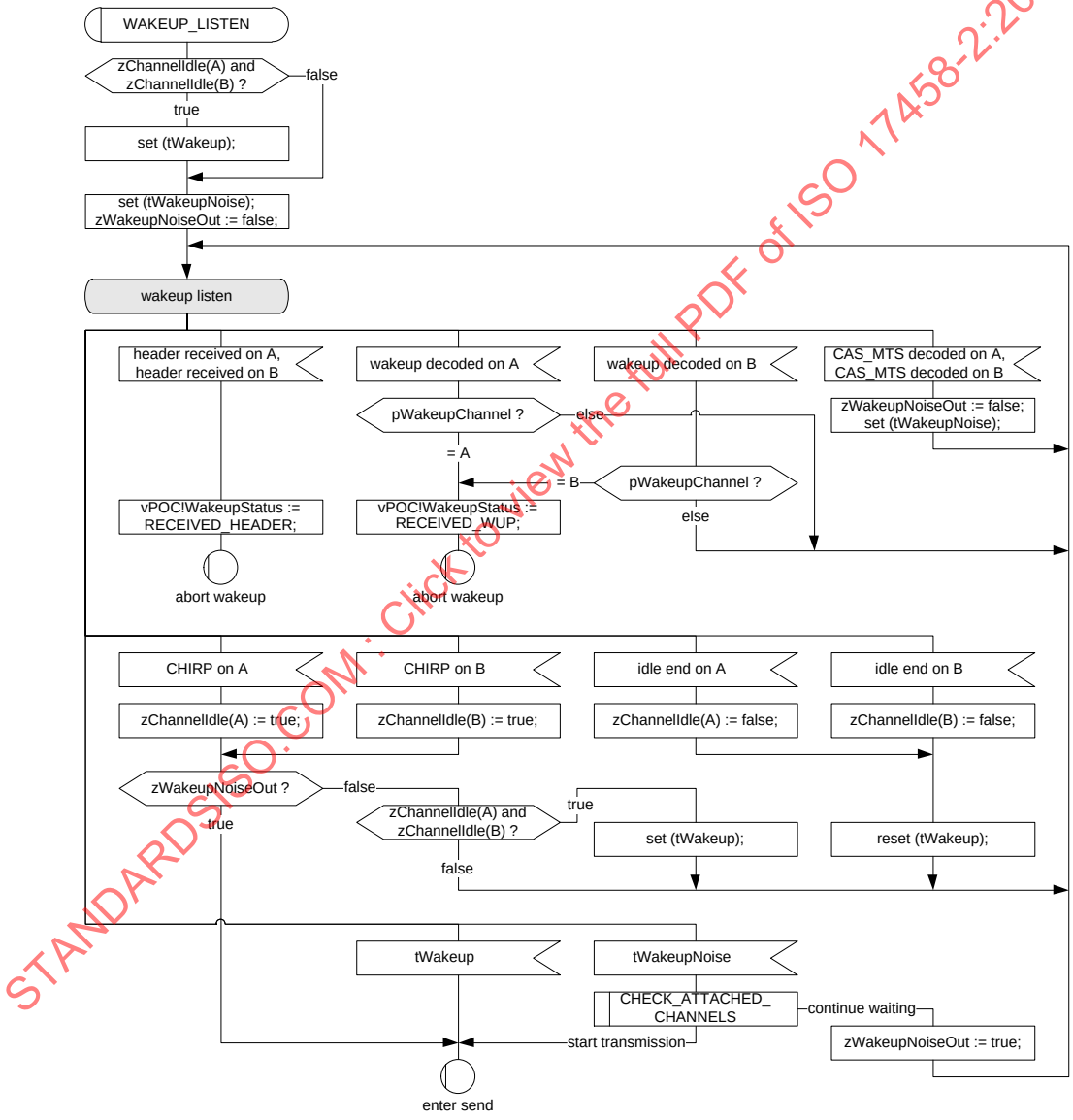



**Figure 131 — Transitions from the POC:wakeup listen state [POC][87)]**

87) If all attached channels are stuck continuously active low POC will remain in the wakeup until the host commands it to a different state.

The 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.

The timer *tWakeup* enables a fast cluster wakeup in a noise free environment, while the timer *tWakeupNoise* enables wakeup under more difficult conditions when noise interference is present or if a single channel is permanently busy.

When ongoing communication is detected or a wakeup of *pWakeupChannel* is already in progress, the wakeup attempt is aborted.

#### 11.2.3.4 The POC:wakeup send state

Figure 132 depicts the Transitions from the *POC:wakeup send* state [POC].

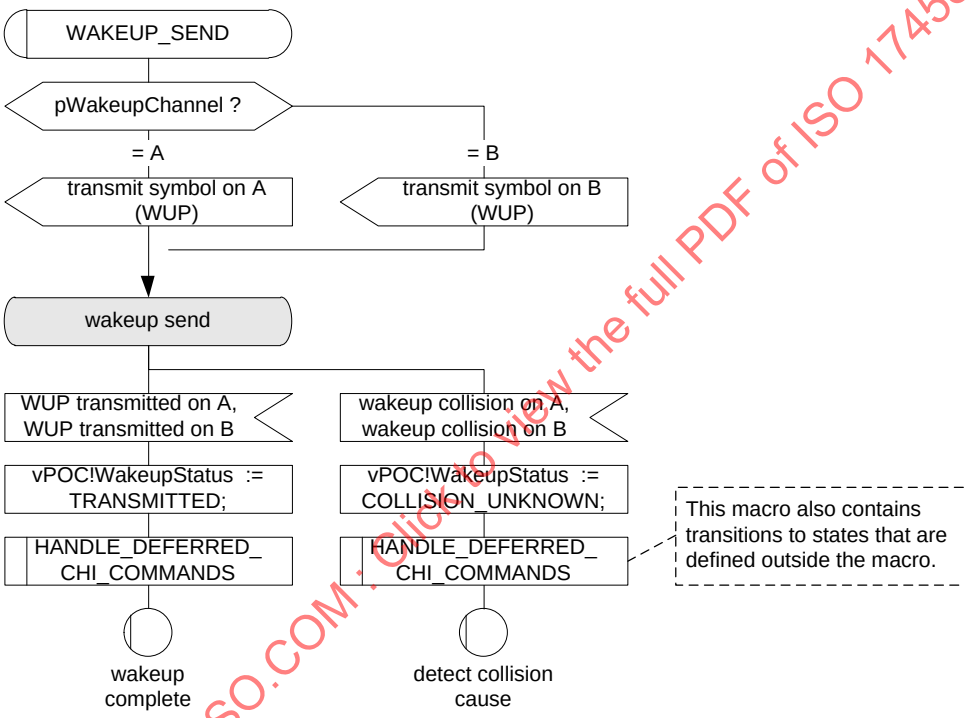



**Figure 132 — Transitions from the POC:wakeup send state [POC]**

In this state, the communication controller transmits the wakeup pattern on the configured channel and checks for collisions.

Since the communication controller transmits the wakeup pattern on *pWakeupChannel*, it cannot really determine whether another node sends a wakeup pattern or frame on this channel during its transmission. Only during the idle portions of the wakeup pattern can it listen to the channel. If during one of these idle portions activity is detected, the communication controller leaves the send phase and enters a succeeding monitoring phase (*POC:wakeup detect* state) so that the cause of the collision might be identified and presented to the host.

#### 11.2.3.5 The POC:wakeup detect state

Figure 133 depicts the transitions from the *POC:wakeup detect* state [POC].

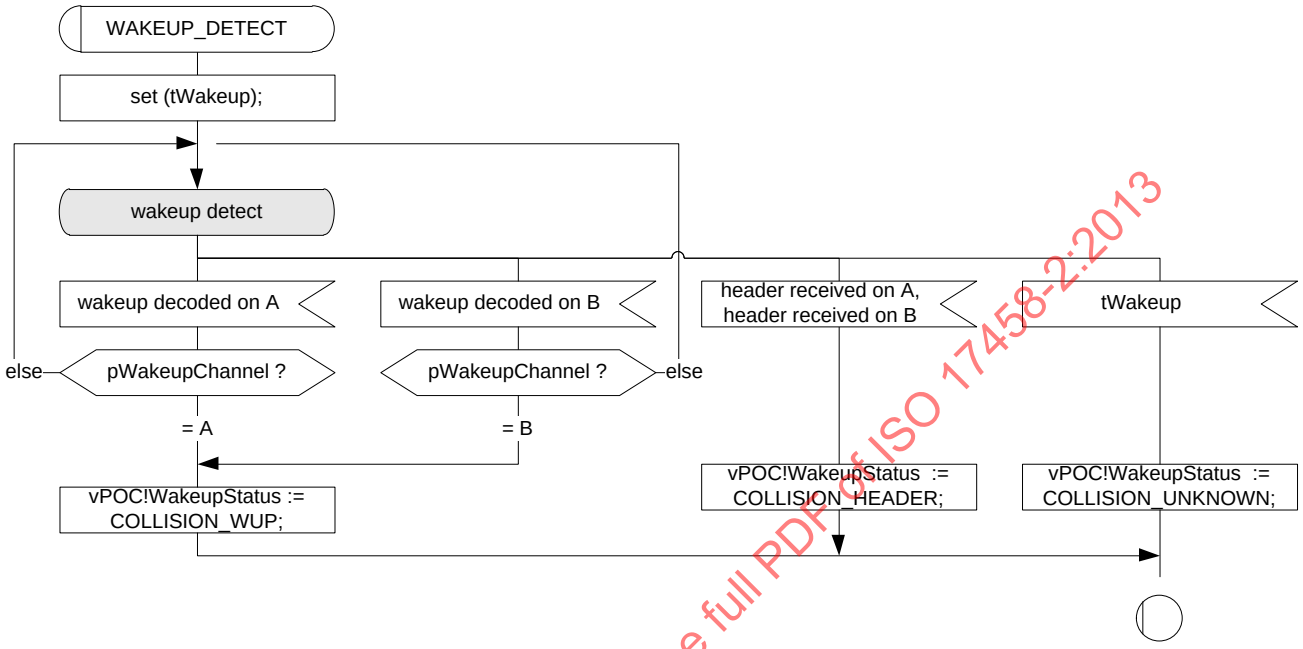



**Figure 133 — Transitions from the POC:wakeup detect state [POC]**

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

There are circumstances where a collision is detected between a WUS and a frame, but the mechanism is not able to identify the actual cause of the collision, returning COLLISION_UNKNOWN instead. If the node that transmitted the frame that collided with the WUS transmits another frame within approximately two cycles the identification procedure will return COLLISION_HEADER, but if the node does not continue transmission the result may still be COLLISION_UNKNOWN. As a result, it is not possible to rely on the existence of a COLLISION_HEADER indication if the cause of a collision was actually a collision with a frame. If COLLISION_HEADER is returned, however, it is certain that frames were present on the network.

This monitoring is bounded by the expiration of the timer *tWakeup*. The detection of either a wakeup pattern indicating a wakeup attempt by another node or the reception of a frame header indicating existing communication causes a direct transition to the *POC:ready* state.

## 11.3 Communication startup and reintegration

### 11.3.1 General

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 synchronisation of all nodes. This strategy is described in the following subclauses.

### 11.3.2 Principles

#### 11.3.2.1 Definition and properties

Before communication startup can be performed, the cluster has to be awake, so the wakeup procedure has to be completed before startup can commence. The startup is performed on all channels synchronously

The action of initiating a startup process is called a coldstart. Only a limited number of nodes may initiate a startup, they are called the coldstart nodes. The process outlined below effectively describes the startup procedure of FlexRay clusters using any synchronisation method. As the TT-E synchronisation method uses a drastically simplified startup process, much of the description is not relevant for TT-E coldstart nodes as indicated in the textual and SDL description.

With the exception of TT-E clusters a coldstart attempt begins with the transmission of a collision avoidance symbol (CAS). Only the coldstart node that transmits the CAS transmits frames in the first four cycles after the CAS. It is then joined first by the other coldstart nodes (if present) and afterwards by all other nodes.

A coldstart node has the configuration parameter *pKeySlotUsedForStartup* set to true and is configured with a frame header containing a startup frame indicator set to one (see clause 8). At least two fault-free coldstart nodes are necessary for the cluster to start up. In TT-D clusters consisting of two nodes, each node shall be a coldstart node. Each startup frame shall also be a sync frame; therefore each coldstart node will also be a sync node. In order to accommodate the possibility of the failure of a coldstart node, a TT-D cluster consisting of three or more nodes should configure at least three nodes as coldstart nodes[88)].

A coldstart node that actively starts the cluster is also called a leading coldstart node. A coldstart node that integrates upon another coldstart node is also called a following coldstart node. By definition a TT-L coldstart node can never be a following coldstart node, as it is the sole coldstart node of its cluster. As TT-E coldstart nodes essentially skip the usual startup process, the distinction is meaningless for them.

A node is in "startup" if its protocol operation control machine is in one of the states contained in the STARTUP macro (*vPOC!State* is set to STARTUP during this time, see Figure 30). 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 C.2).

#### 11.3.2.2 Principle of operation

##### 11.3.2.2.1 General

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.

##### 11.3.2.2.2 Startup performed by the coldstart nodes

During startup a coldstart node behaves as follows.

— Only the coldstart nodes can initiate the cluster start up.

— When the TT-D synchronisation method is used by the cluster, each of the TT-D coldstart nodes finishes its startup as soon as stable communication with one of the other coldstart nodes is established. A TT-L coldstart node always finishes the startup after six cycles. A TT-E coldstart node finishes its startup as soon as it receives the first cycle start from its time gateway source node after having previously received information about rate and offset correction terms and confirmation that the time gateway source node is in *POC:normal active*.

88) This does not imply any restrictions concerning which nodes may initiate a cluster wakeup.

#### 11.3.2.2.3 Integration of the non-coldstart nodes

During startup a non-coldstart node behaves as follows.

— A non-coldstart node requires at least two startup frames with different frame IDs for integration. In a TT-D cluster this condition ensures that each non-coldstart node always joins the majority of the coldstart nodes[89)].

— Integrating non-coldstart nodes may start their integration before coldstart nodes have finished their startup.

— Integrating non-coldstart nodes in a TT-D cluster will not finish their startup until at least two coldstart nodes have finished their startup.

### 11.3.3 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 134.

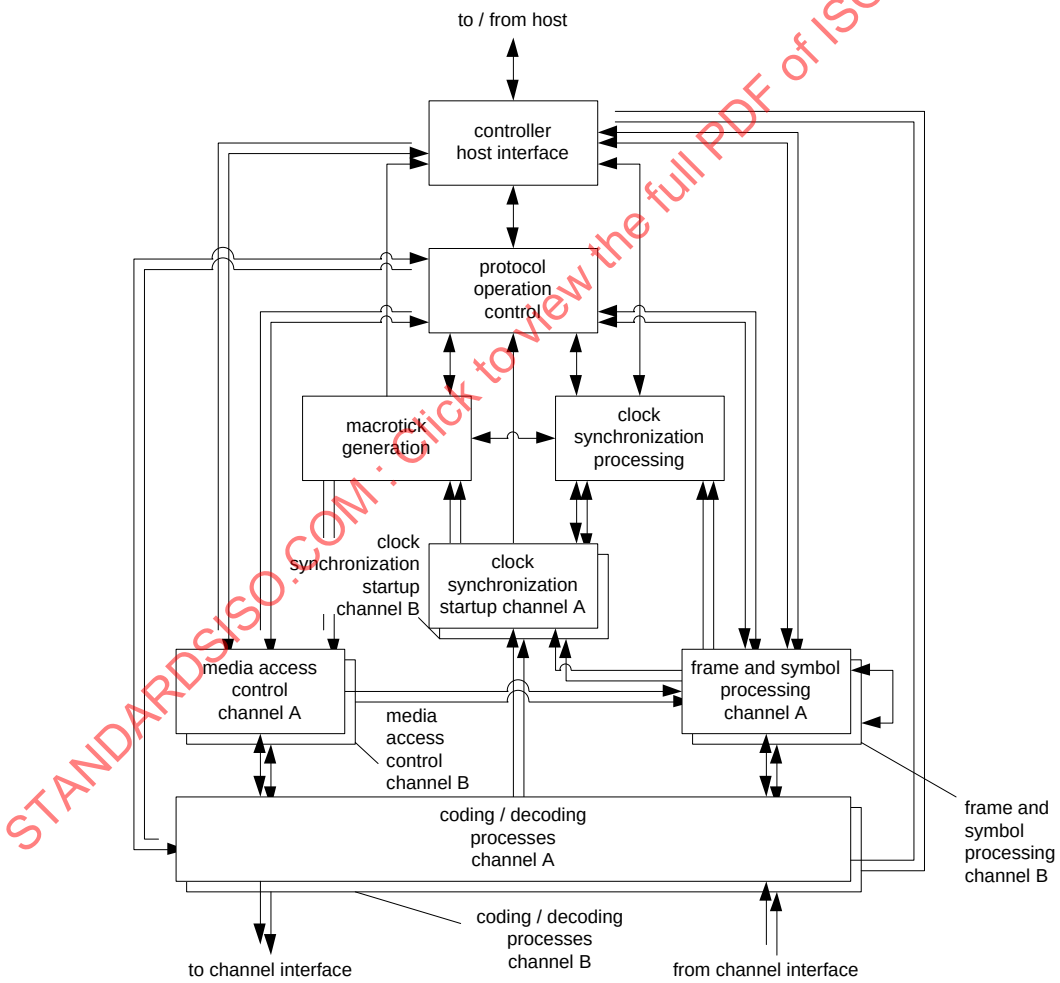



**Figure 134 — Protocol operation control context**

89) This is the case under the assumption that the cluster contains exactly the three recommended coldstart nodes.

### 11.3.4 Coldstart inhibit mode

In coldstart inhibit mode the communication controller is prevented from initializing the TDMA communication schedule. The communication controller is automatically set into coldstart inhibit mode prior to entering the *POC:ready* state. The CC remains in this mode until the host commands the node to leave coldstart inhibit mode by means of an ALLOW_COLDSTART command issued after the POC enters the *POC:ready* state[90)].

While in coldstart inhibit mode, the communication controller is not allowed to assume the role of a leading coldstart node, i.e., entering the coldstart path is prohibited. The node is still allowed to integrate into a running cluster or to act as a following coldstart node, transmitting startup frames after another coldstart node has started the initialisation of cluster communication. Once the node is synchronized and integrated into cluster communication, the coldstart inhibit mode does not restrict the node's ability to transmit frames.

The coldstart inhibit status of a node is represented by the *vColdstartInhibit* variable, with a value of true indicating the node is in coldstart inhibit mode and a value of false indicating the node is not in coldstart inhibit mode.

The coldstart inhibit mode can be used either to completely prohibit active startup attempts of a node (if the host never issues an ALLOW_COLDSTART command), or only delay them (if the host issues the CHI command during a startup procedure). As a result, the coldstart inhibit mode may be used to attempt to ensure that all fault-free coldstart nodes are ready for startup and in the *POC:coldstart listen* state before one of them initiates a coldstart attempt.

A coldstart node in a TT-E cluster (i.e., a node with *pExternalSync* set to true) will never be in coldstart inhibit mode as it follows a fundamentally different path for startup. Such nodes will operate as a coldstart node even without the issuance of an ALLOW_COLDSTART command.

### 11.3.5 Startup state diagram

#### 11.3.5.1 Overview of the different startup paths

There are several ways that a node can enter communication. Subclause 11.3.5.2 describes the path followed by the leading coldstarter in a TT-D or TT-L cluster. Subclause 11.3.5.3 describes the paths available to a TT-D coldstart node that does not act as the leading coldstarter. Subclause 11.3.5.5 describes the path followed by a TT-E coldstart node. Subclause 11.3.5.6 describes the path of a non-coldstart node in all cluster types.

All of these subclauses provide only an overview[91)] of the operation - the precise behaviour is defined by the SDL descriptions in the subsequent clauses.



90) There is no mechanism that allows the host to explicitly place the node into coldstart inhibit mode - this happens automatically as part of the process of reaching *POC:ready*.

91) There are a large number of paths possible involving multiple executions of the STARTUP_PREPARE macro (for example, paths involving one or more aborted startup attempts). No overview of these paths is provided, but their behaviour is explicitly defined by the SDL descriptions in this subclause.

Figure 135 depicts the startup state diagram [POC].

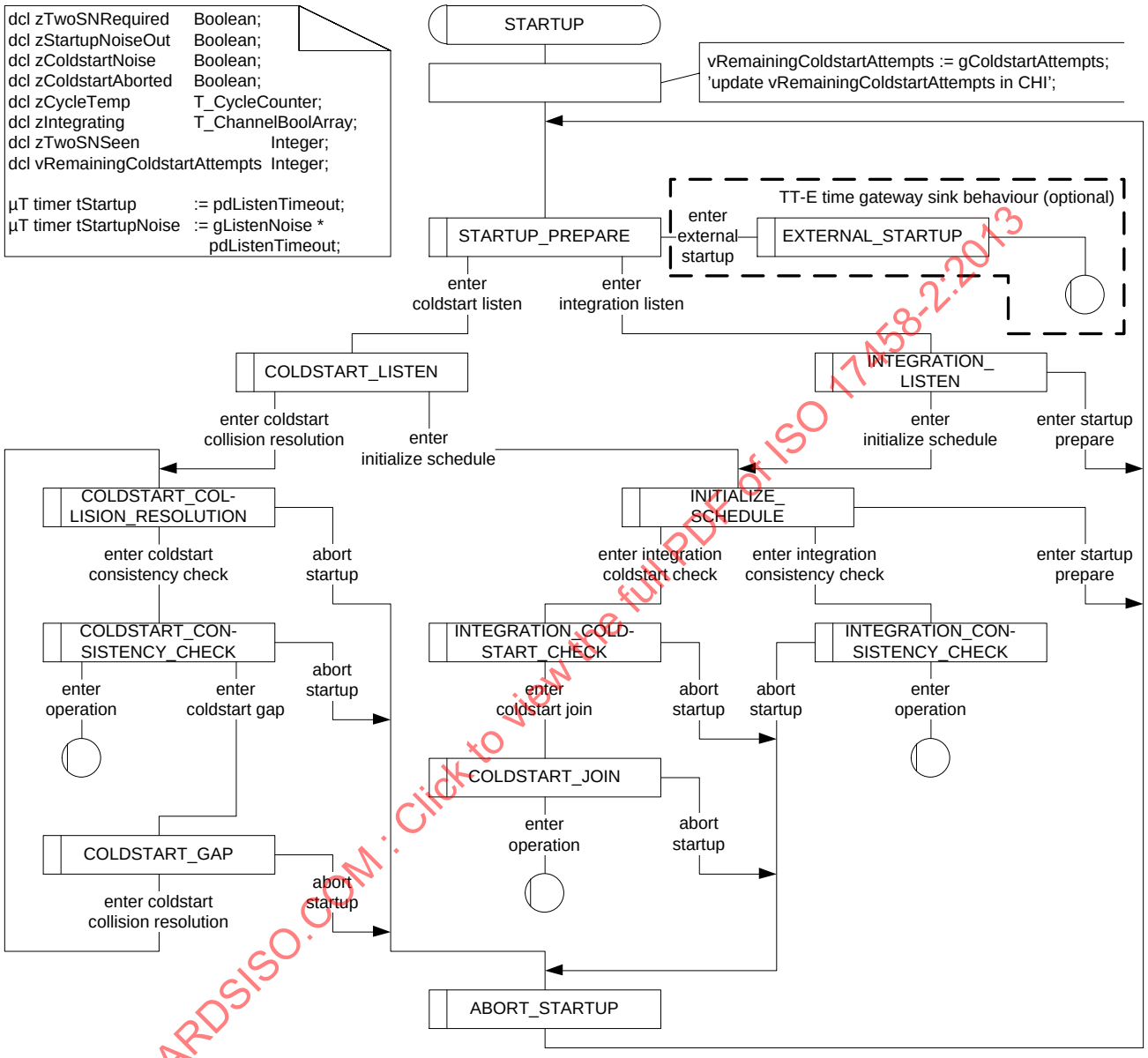



**Figure 135 — Startup state diagram [POC]**

Figure 136 depicts the helpful macros for startup [POC].

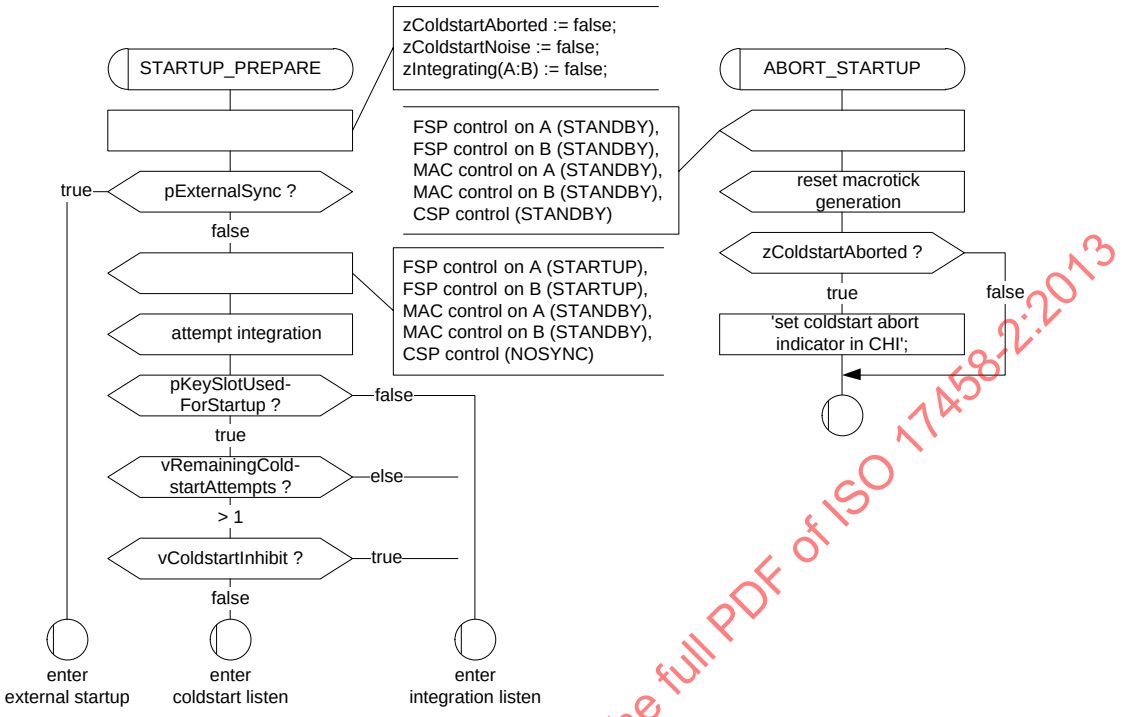



**Figure 136 — Helpful macros for startup [POC]**

Figure 137 depicts the example of state transitions for a fault-free startup.

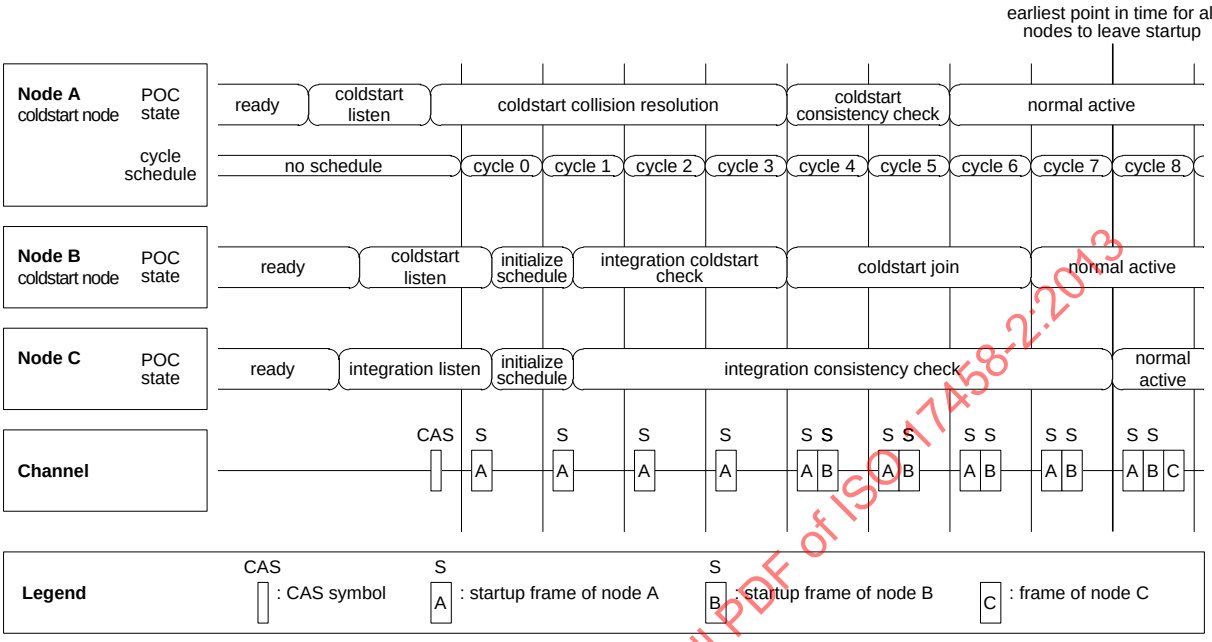



**Figure 137 — Example of state transitions for a fault-free startup[92)]**

### 11.3.5.2 Path of a TT-D leading coldstart node

Node A in Figure 137 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 142).

If no communication[93)] 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 144).

In cycle four, other coldstart nodes begin to transmit their startup frames. The node that initiated the coldstart collects all startup frames from cycle four and five and performs the clock correction as described in clause 12. If clock correction does not signal any errors and the node has received at least one valid startup

92) 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 clause 12).
93) See Figure 142 for exact definition.

frame pair, the node leaves startup and enters operation (see SDL macro COLDSTART_CONSISTENCY_CHECK in Figure 145).

#### 11.3.5.3 Path of a TT-D following coldstart node

Node B in Figure 137 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 142).

If communication[94] is received, it tries to integrate to a transmitting coldstart node[95]. It tries to receive a valid pair of startup frames to derive its schedule and clock correction from the coldstart node (see clause 12 and see SDL macro INITIALIZE_SCHEDULE in Figure 147).

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 148).

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 149).

Another path, not shown in Figure 137, is also possible for an integrating coldstart node. If, at the time of the execution of the STARTUP_PREPARE macro, the node is prevented from acting as a leading coldstarter (either because the *vColdstartInhibit* flag is set to true, or because the *vRemainingColdstartAttempts* variable indicates there are no remaining coldstart attempts) the node will instead begin to act as a normal integrating node, waiting for a leading coldstarter to begin transmissions that will initialize the schedule (see SDL macro INTEGRATION_LISTEN). Once such communication is detected, the node then executes the INITIALIZE_SCHEDULE macro and behaves as described earlier in this subclause.

#### 11.3.5.4 Path of a TT-L coldstart node

Node A in Figure 138 follows this path.

When a TT-L coldstart node enters startup, it listens to its attached channels and attempts to receive FlexRay frames (see SDL macro COLDSTART_LISTEN in Figure 142) even though it is the sole provider of startup frames of its cluster. As no communication can be received, the node soon commences a coldstart attempt. The initial transmission of a CAS symbol is succeeded by the first regular cycle, which is given a cycle number of zero.



94) See clause 12 for exact definition.
95) Presumably it is the node that initiated the coldstart, but not necessarily.

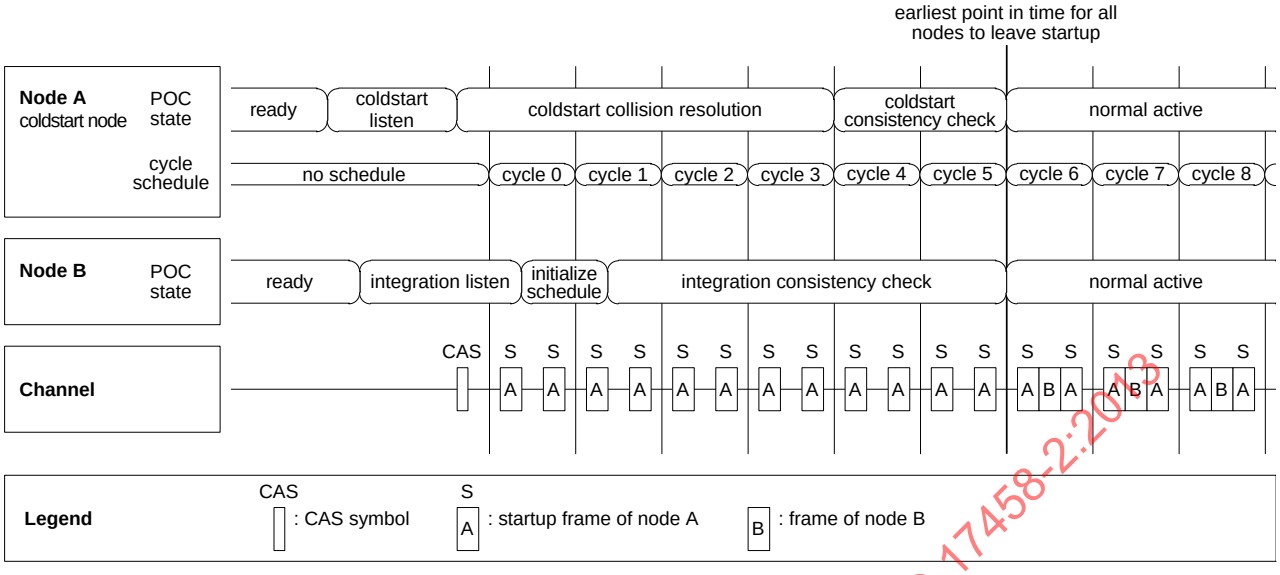



**Figure 138 — Example of state transitions for a fault-free startup in a TT-L cluster[96)]**

From cycle zero on, the node transmits its two startup frames.

As no collision with startup frames or CAS symbols transmitted by other nodes can occur (in the fault-free case), the TT-L coldstart node proceeds straight through the *POC:coldstart collision resolution* state. In the following two cycles, the TT-L coldstart node remains in *POC:coldstart consistency check* and always proceeds on to *POC:normal active*, as the two startup frames it provides are sufficient to initialize and maintain the TT-L cluster (see SDL macro COLDSTART_CONSISTENCY_CHECK in Figure 145).

### 11.3.5.5 Path of a TT-E coldstart node

Figure 139 depicts a system topology that is used to describe the path followed by TT-E coldstart nodes. In this figure, which depicts two independent time gateways, nodes M, N, P and Q are connected to the time source cluster (a cluster using the TT-D synchronisation mode) and nodes A, B, and C are connected to the time sink cluster. In the TT-D cluster, nodes N, P, and Q are coldstart nodes.

Figure 140 depicts the path of nodes A and B, the coldstart nodes of the TT-E cluster. As Figure 141 shows, the behaviour of a node following this path is straightforward.



96) 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 clause 12).

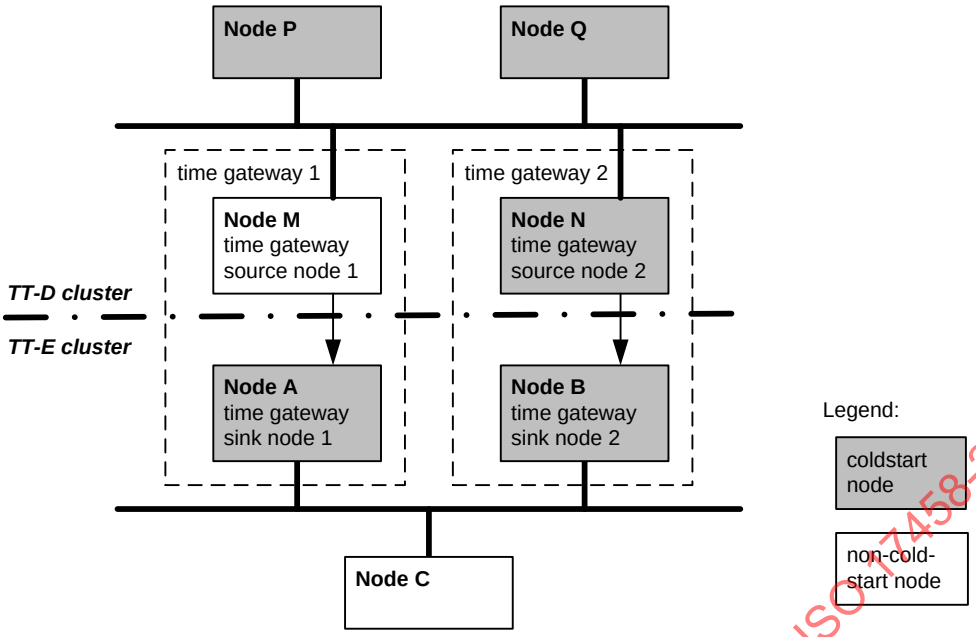



**Figure 139 — Topology for the startup example of a TT-E cluster**

The node first starts the CSP, MAC and FSP processes and then awaits the first *cycle start* signal; receiving this, the node directly continues into *POC:normal active*. The *cycle start* signal is generated as soon as the (just started) CSP process has managed to synchronize itself onto the time gateway source (see Figure 156).

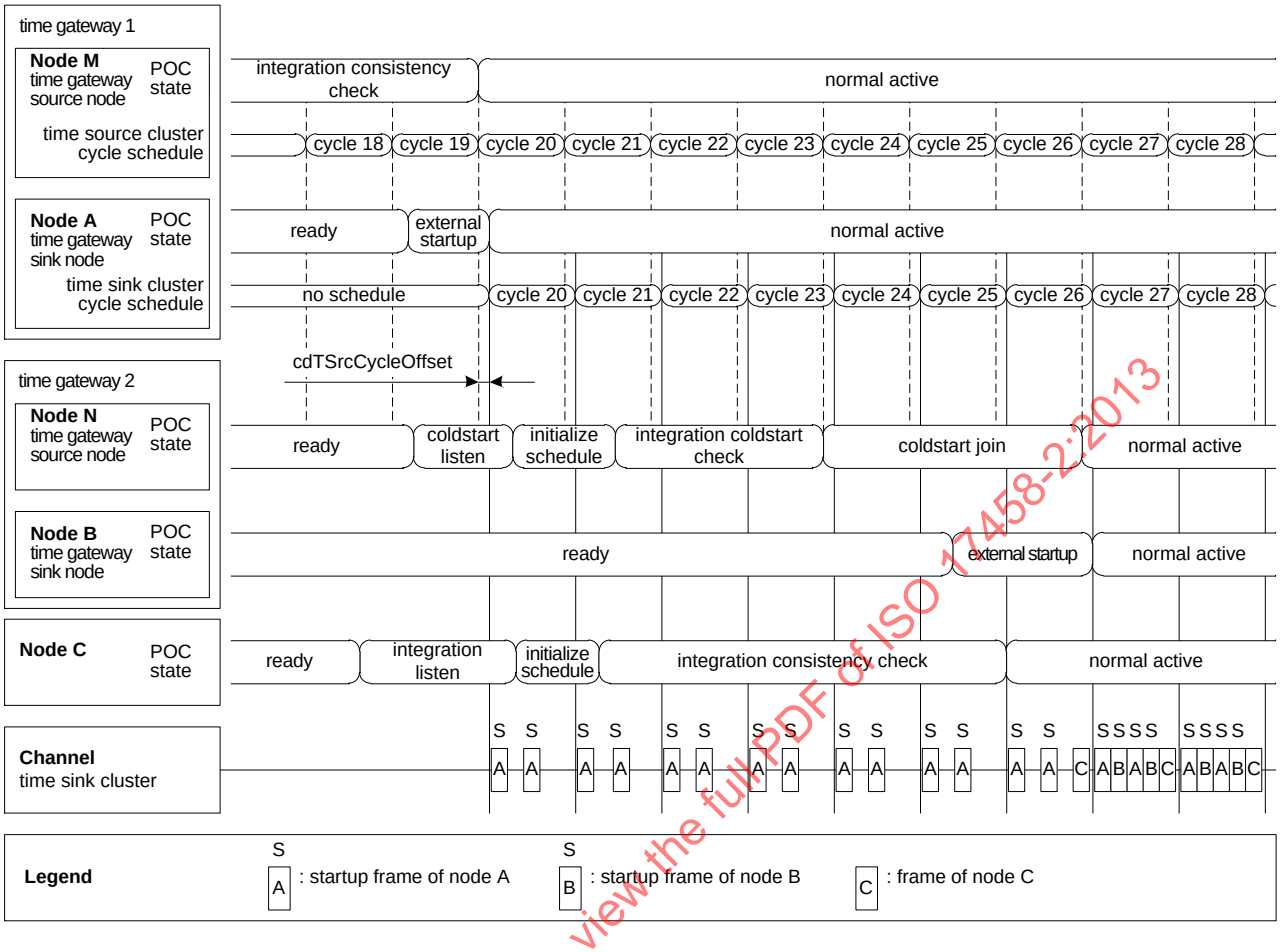



**Figure 140 — Example of state transitions for a fault-free startup in a TT-E cluster[97)]**

Node A in Figure 140 enters *POC:normal active* very shortly after its time gateway source node (node M) has entered *POC:normal active* itself. The time gateway source node has provided the node A with all relevant information about clock correction values during cycles 18 and 19, but as node M was not yet in *POC:normal active*, node A could not proceed. Node A uses the cycle counter value provided by the time source gateway node and therefore directly starts with the cycle number 20, the same cycle number the time source cluster currently uses. The cycle schedules of Figure 140 have been drawn slightly offset to one another to symbolize the fixed offset of *cdTSrcCycleOffset* microticks between the cluster schedules.

The second TT-E coldstart node, node B, also enters *POC:normal active* as soon as it has received all relevant terms from its time gateway source node (node N) and does not verify its view on the schedule against the already present startup frames. Node B is not required for node C to complete its startup.

97) 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 clause 12).

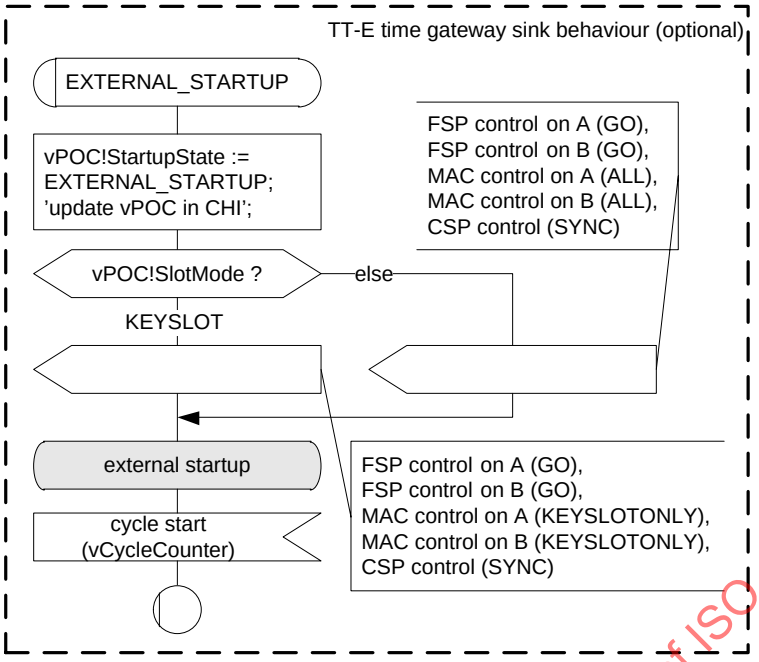



**Figure 141 — External startup state [POC]**



#### 11.3.5.6 Path of a non-coldstart node

Node C in Figure 137, Node B in Figure 138, and Node C in Figure 140 follow this path.

When a non-coldstart node enters startup, it listens to its attached channels and tries to receive FlexRay frames (see SDL macro INTEGRATION_LISTEN in Figure 150).

If communication[98)] 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 clause 12 and see SDL macro INITIALIZE_SCHEDULE in Figure 147).

In the following double cycles, it tries to find at least two startup frames that fit into its own schedule. In a TT-D cluster these frames will come from different coldstart nodes. If this fails, or if clock correction signals an error, the node aborts the integration attempt and tries again.

After receiving two valid startup frame pairs with different frame IDs for two consecutive double cycles, the node leaves startup and enters operation.

For TT-D clusters, this means that the non-coldstart node leaves startup at least two cycles later than the node that initiated the coldstart. Effectively, all nodes of a TT-D cluster can leave startup at the end of cycle 7, just before entering cycle 8 (see Figure 137 and SDL macro INTEGRATION_CONSISTENCY_CHECK in Figure 151). For TT-D and TT-E clusters, this time is reduced by one double cycle, as the two necessary startup frames are present as much earlier (see Figure 138 and Figure 140).

98) See clause 12 for exact definition.

### 11.3.5.7 The POC:coldstart listen state

Figure 142 depicts the Transitions from the *POC:coldstart listen* state [POC].

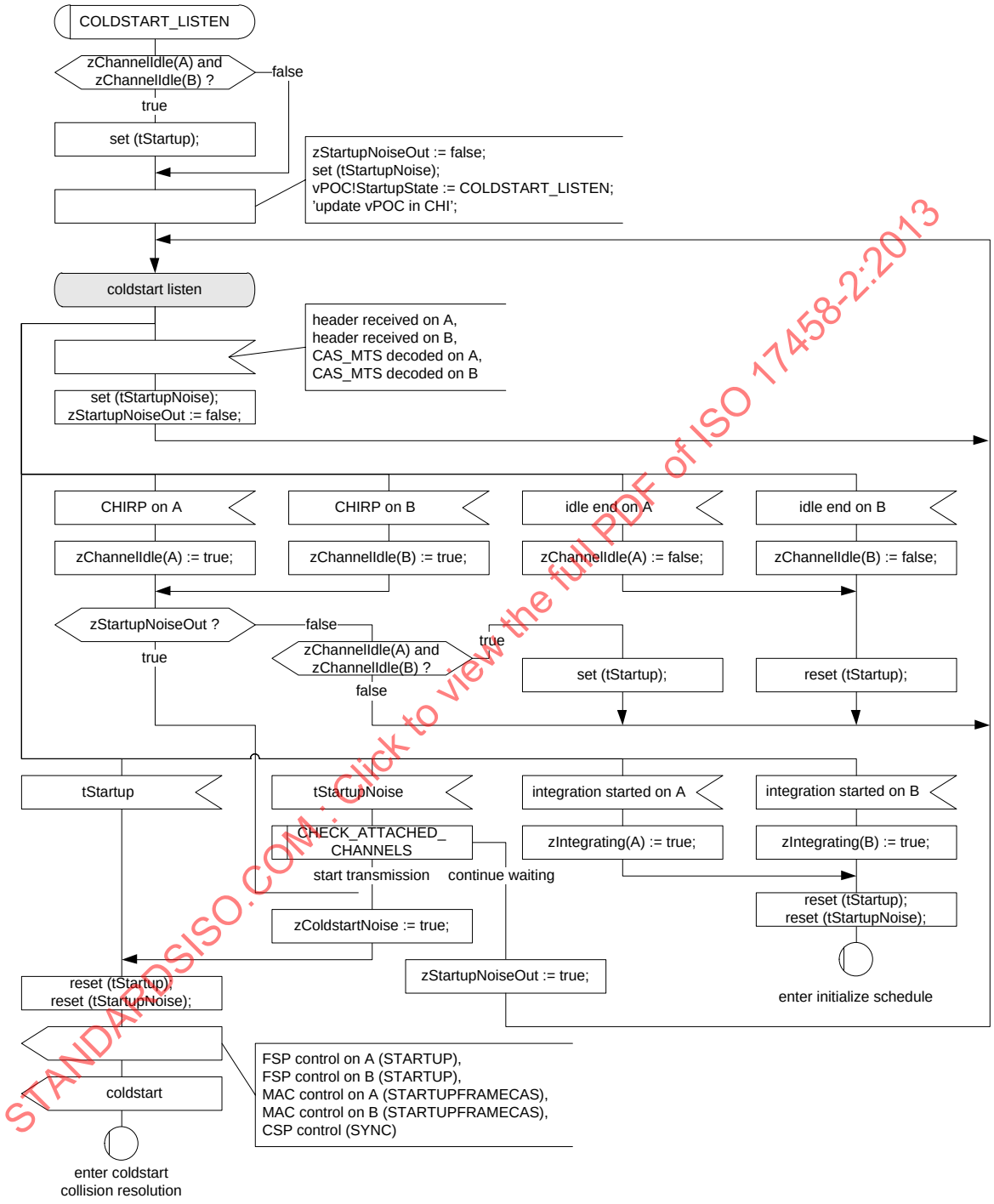





**Figure 142 — Transitions from the POC:coldstart listen state [POC][99)]**

99) If all attached channels are stuck continuously active low POC will remain in the startup until the host commands it to a different state.

Figure 143 depicts the Macro CHECK_ATTACHED_CHANNELS [POC].

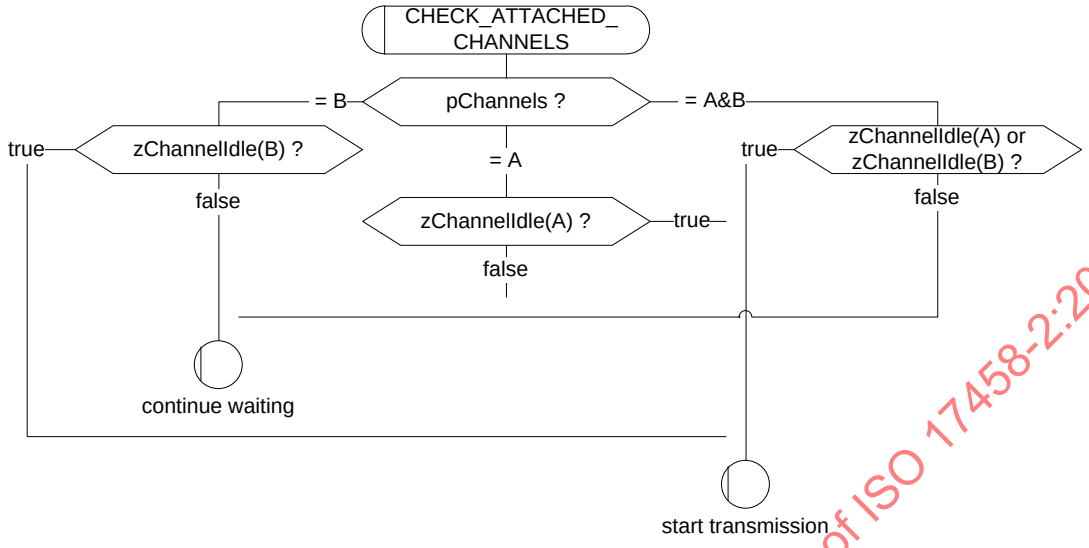



**Figure 143 — Macro CHECK_ATTACHED_CHANNELS [POC]**

A coldstart node still allowed[100)] 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 clause 12 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 clause 7 for a description of channel states). It is restarted when all attached channels are in idle state. The timer *tStartupNoise* is only restarted by the reception of correctly decoded headers or CAS symbols to guarantee a cluster startup when noise interference is present or if a single channel is permanently busy.



100) See macro STARTUP_PREPARE in Figure 136. The condition *'vRemainingColdstartAttempts* > 1' arises from the necessity of using up one round through the *POC:coldstart collision resolution* state for the collision resolution and needing the second round for actually integrating the other nodes.

### 11.3.5.8 The POC:coldstart collision resolution state

Figure 144 depicts the transitions from the *POC:coldstart collision resolution* state [POC].

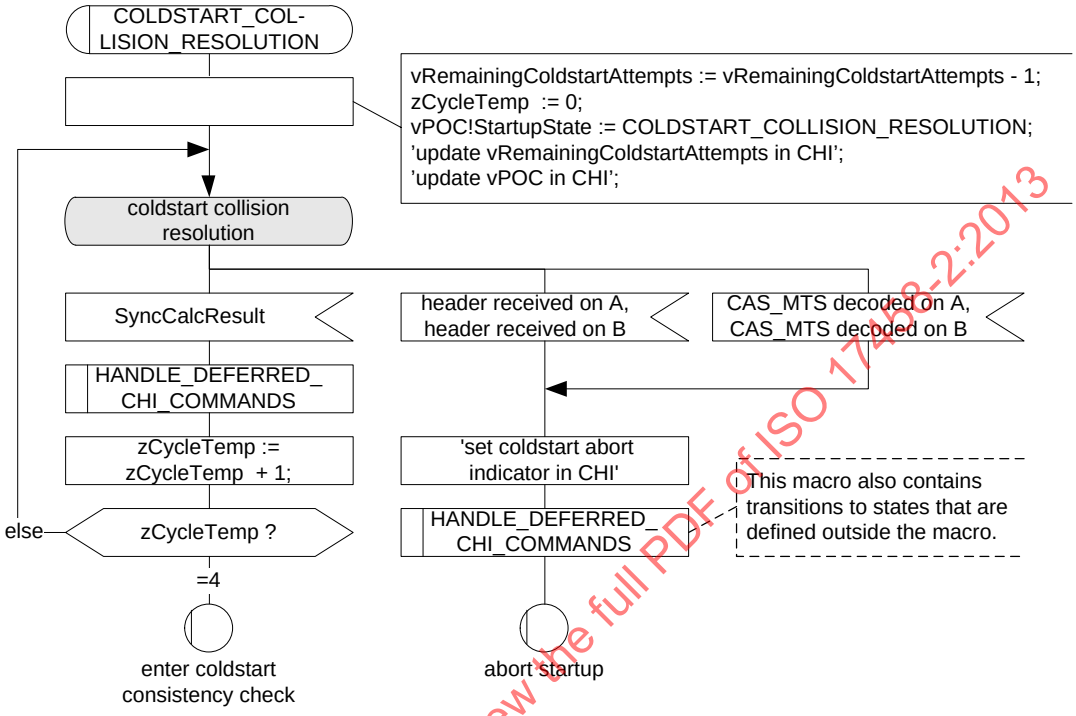



**Figure 144 — Transitions from the POC:coldstart collision resolution state [POC]**



The purpose of this state is to detect and resolve collisions between multiple simultaneous coldstart attempts of several coldstart nodes. Each entry into this state starts a new coldstart attempt by this node.

The reception of a complete header without coding errors or the reception of a valid CAS symbol causes the communication controller to abort the coldstart attempt. This resolves conflicts between multiple coldstart nodes performing a coldstart attempt at the same time, so only one leading coldstart node remains.

In the fault-free case and under certain configuration constraints (see Annex A) only one coldstart node will proceed to the *POC:coldstart consistency* check state. The other nodes abort startup since they received a frame header from the successful coldstart node.

The number of coldstart attempts that a node is allowed to make is restricted to the initial value of the variable *vRemainingColdstartAttempts*. *vRemainingColdstartAttempts* is reduced by one for each attempted coldstart. A node may enter the *POC:coldstart listen* state only if this variable is larger than one and it may enter the *POC:coldstart collision resolution* state only if this variable is larger than zero. A value of larger than one is required for entering the *POC:coldstart listen* state because one coldstart attempt may be used for performing the collision resolution, in which case the coldstart attempt could fail.

After four cycles in this state, the node enters the *POC:coldstart consistency check* state.

### 11.3.5.9 The POC:coldstart consistency check state

Figure 145 depicts the transitions from the *POC:coldstart consistency check* state [POC].

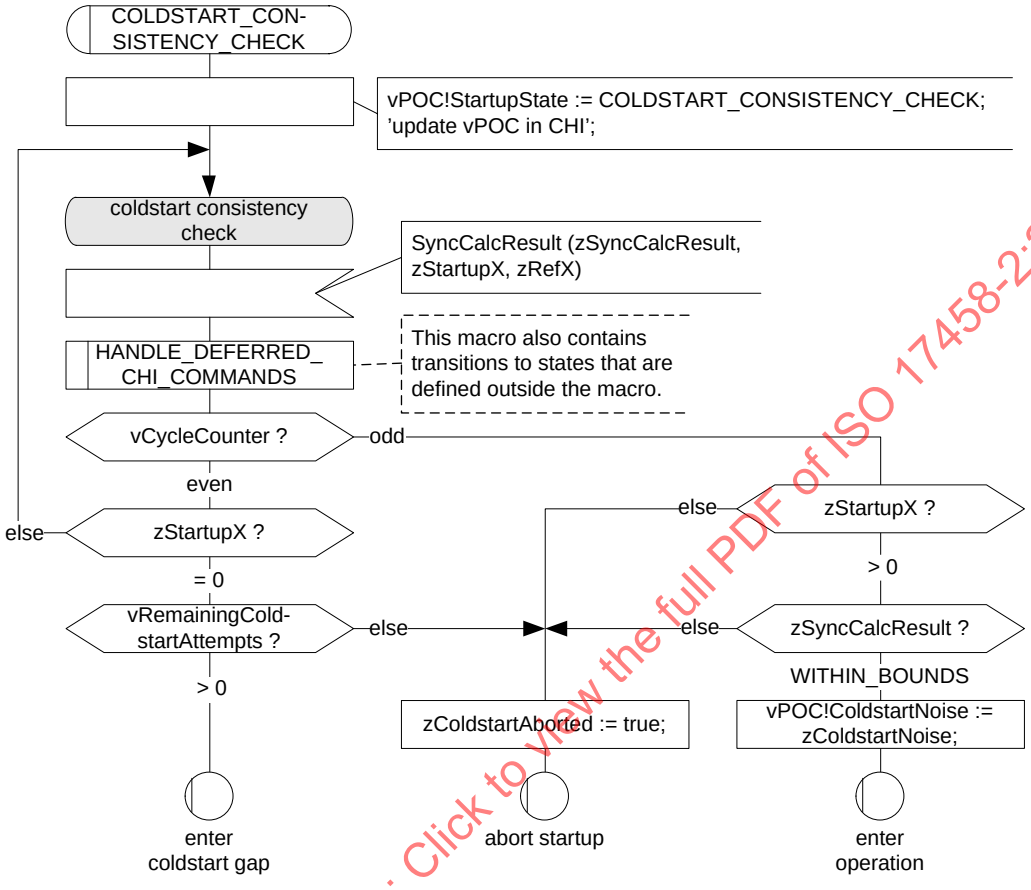



**Figure 145 — Transitions from the POC:coldstart consistency check state [POC][101)]**



In this state, the leading coldstart node checks whether the frames transmitted by other following coldstart nodes (non-coldstart nodes cannot yet transmit in the fault-free case) fit into its schedule.

If a TT-D coldstart node receives no valid startup frames in the even cycle in this state, it is assumed that the other coldstart nodes were not ready soon enough to initialize their schedule from the first two startup frames sent during the *POC:coldstart collision resolution* state. Therefore, if another coldstart attempt is allowed, the node enters the *POC:coldstart gap* state to wait for the other coldstart nodes to get ready.

If a TT-D coldstart node has received a valid startup frame in the even cycle in this state, but the clock correction signals errors in the odd cycle or no valid pair of startup frames can be received in the double cycle, the node aborts the coldstart attempt.

If a TT-D coldstart node has received a valid pair of startup frames and the clock correction signals no errors the node leaves startup and enters operation (see clause 6).

---

101) *zStartupX* is *zStartupNodes* in even cycles and *zRxStartupPairs* in odd cycles. *zRefX* is *zRefNode* in even cycles and *zRefPair* in odd cycles. See Figure 157 for details.

A TT-L coldstart node will enter operation after having remained for two cycles in this state, as it is the sole provider of startup frames of the cluster.

#### 11.3.5.10 The POC:coldstart gap state

Figure 146 depicts the transitions from the *POC:coldstart gap* state [POC].

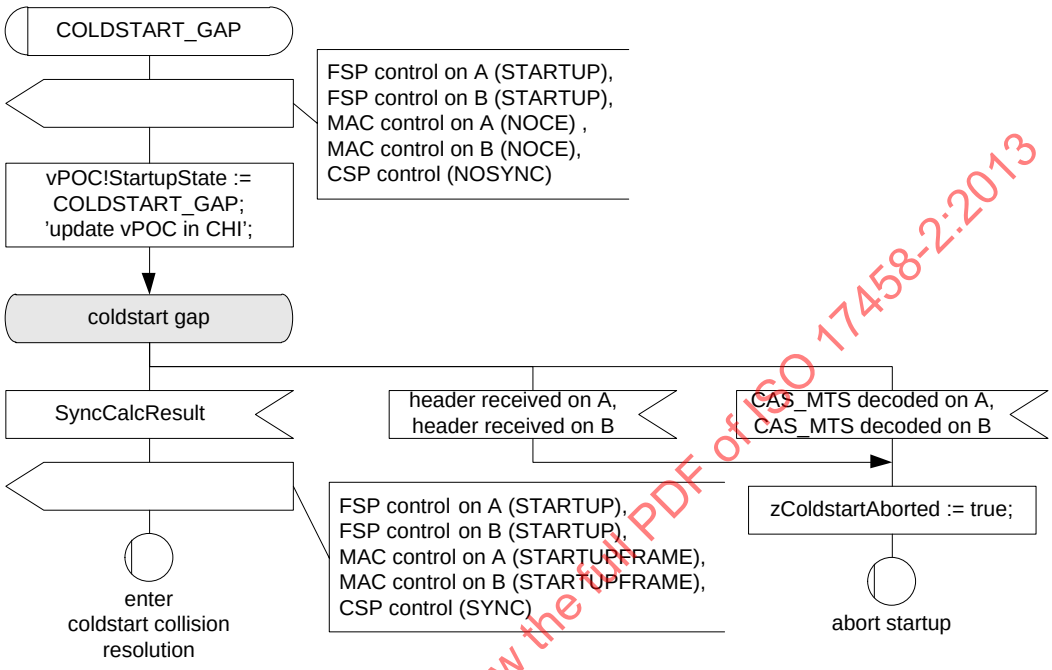



**Figure 146 — Transitions from the POC:coldstart gap state [POC]**

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.

### 11.3.5.11 The POC:initialize schedule state

Figure 147 depicts the transitions from the *POC:initialize schedule* state [POC].

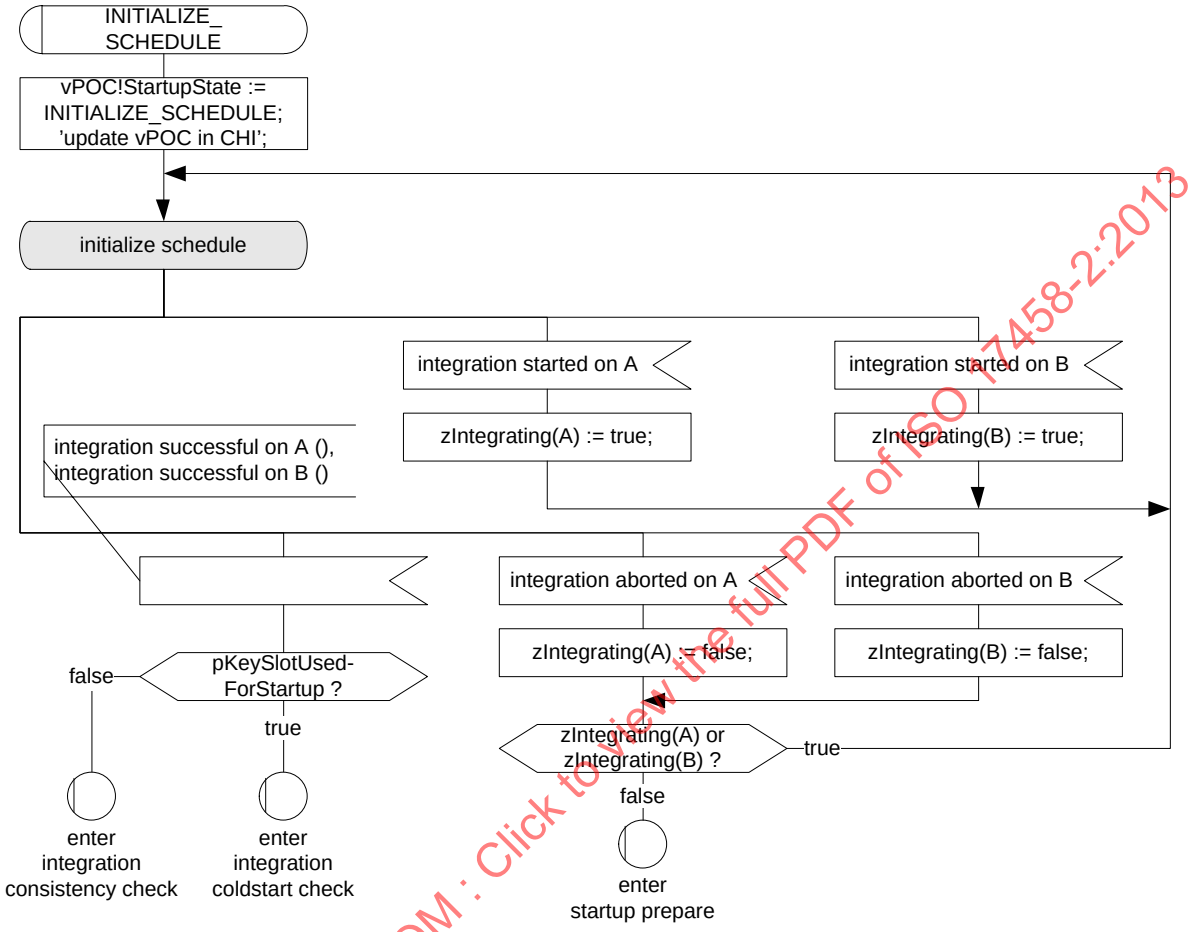



**Figure 147 — Transitions from the POC:initialize schedule state [POC]**

As soon as a valid startup frame has been received in one of the listen states (see Figure 135), the *POC:initialize* schedule state is entered. If clock synchronisation successfully receives a matching second valid startup frame and derives a schedule from them (indicated by receiving the signal *integration successful on A* or *integration successful on B*), the POC goes to the *POC:integration coldstart check* state (for coldstart nodes) or the *POC:integration consistency check* state (for non-coldstart nodes).

### 11.3.5.12 The POC:integration coldstart check state

Figure 148 depicts the transitions from the *POC:integration coldstart check* state [POC].

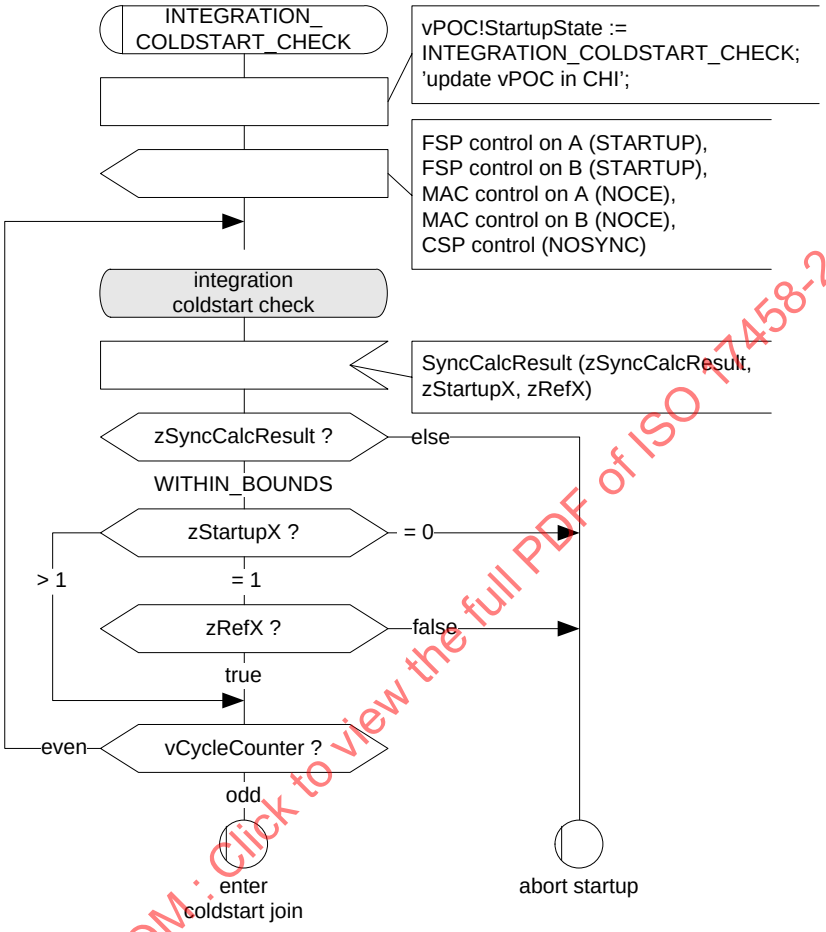



**Figure 148 — Transitions from the POC:integration coldstart check state [POC][102)]**

Only integrating (following) coldstart nodes pass through this state. In this state it shall be verified that the clock correction can be performed correctly, that at least one coldstart node is still available, and if exactly one coldstart node is available that it is the same coldstart node that was used to initialize the schedule.

The clock correction is activated and if any error is signalled 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 shall 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.

102) *zStartupX* is *zStartupNodes* in even cycles and *zRxStartupPairs* in odd cycles. *zRefX* is *zRefNode* in even cycles and *zRefPair* in odd cycles. See Figure 157 for details.

### 11.3.5.13 The POC:coldstart join state

Figure 149 depicts the transitions from the *POC:coldstart join* state [POC].

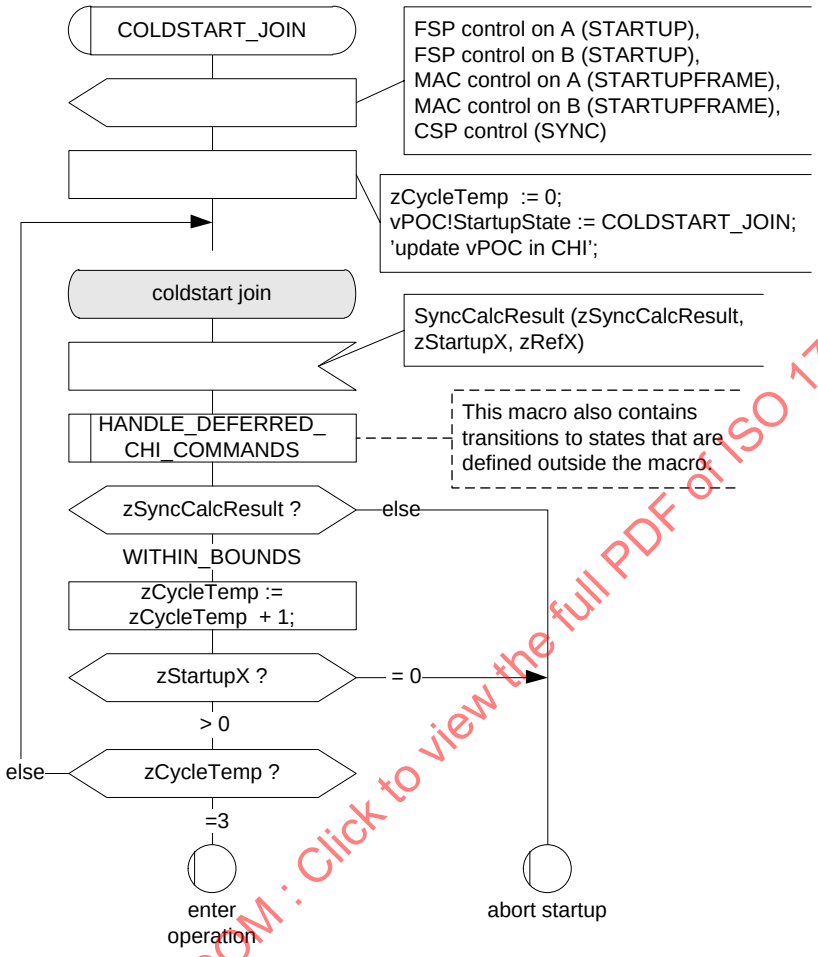



**Figure 149 — Transitions from the POC:coldstart join state [POC][103)]**

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 clause 6).

103) *zStartupX* is *zStartupNodes* in even cycles and *zRxStartupPairs* in odd cycles. *zRefX* is *zRefNode* in even cycles and *zRefPair* in odd cycles. See Figure 157 for details.

#### 11.3.5.14 The POC:integration listen state

Figure 150 depicts the transitions from the *POC:integration listen* state [POC].

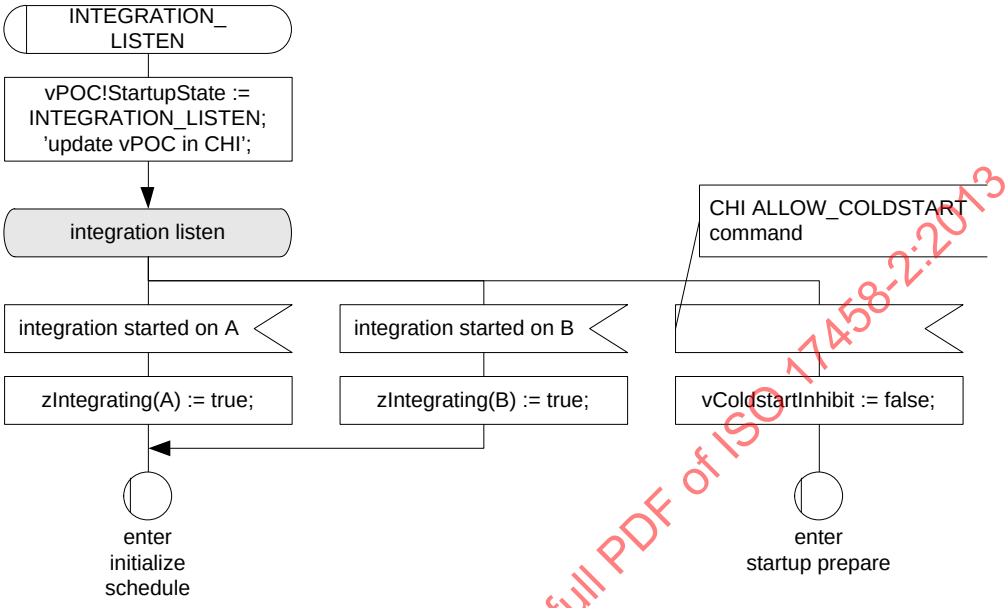



**Figure 150 — Transitions from the POC:integration listen state [POC]**

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

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

### 11.3.5.15 The POC:integration consistency check state

Figure 151 depicts the transitions from the *POC:integration consistency check* state [POC].

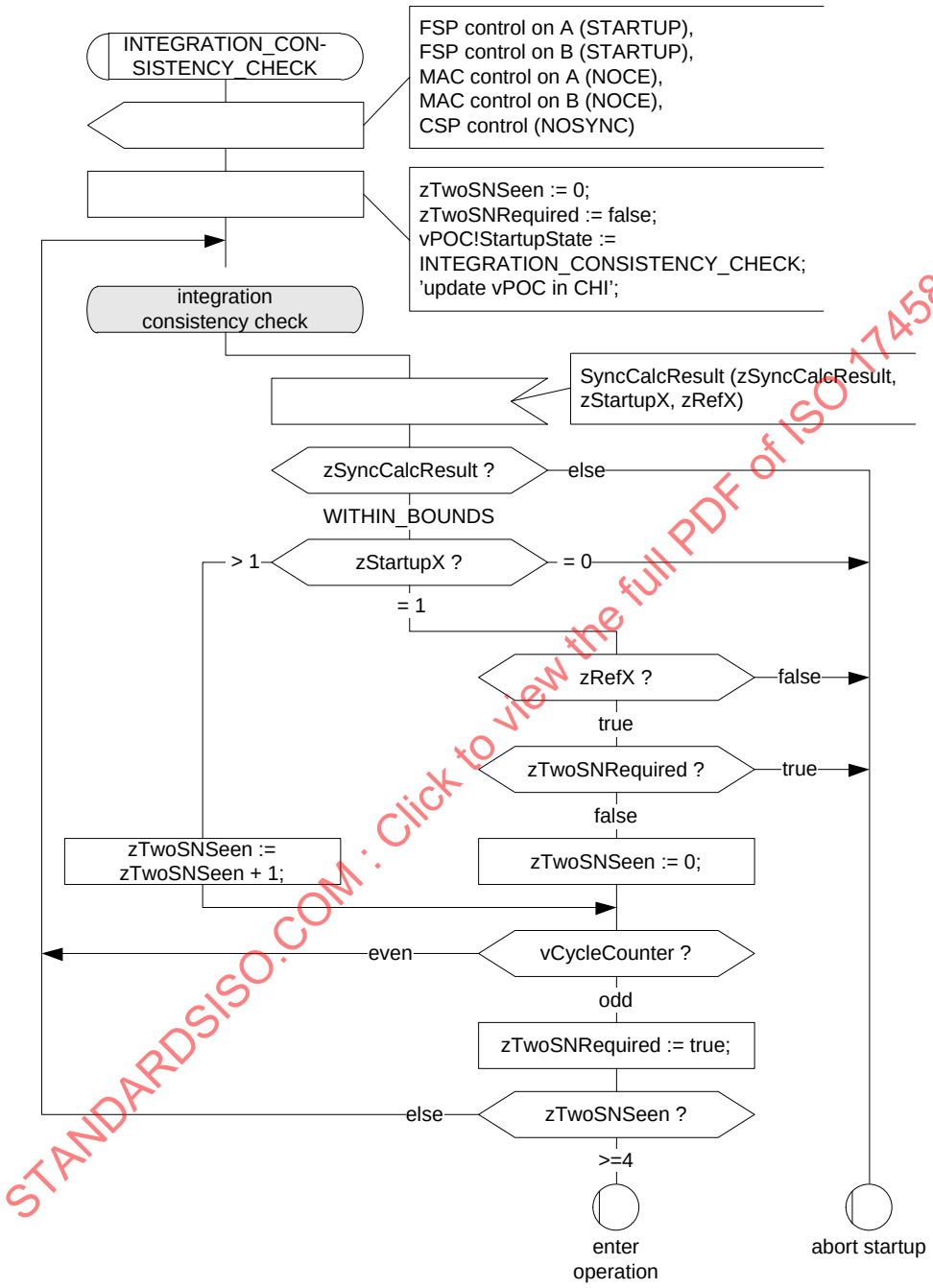





**Figure 151 — Transitions from the POC:integration consistency check state [POC][104)]**

104) *zStartupX* is *zStartupNodes* in even cycles and *zRxStartupPairs* in odd cycles. *zRefX* is *zRefNode* in even cycles and *zRefPair* in odd cycles. See Figure 157 for details.

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 signalled 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 shall 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 shall 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 clause 6). 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.

# 12 Clock synchronisation

## 12.1 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 (i.e. oscillator), the internal time bases of the various nodes diverge after a short time, even if all the internal time bases are initially synchronized.

Figure 152 depicts the clock synchronisation context.

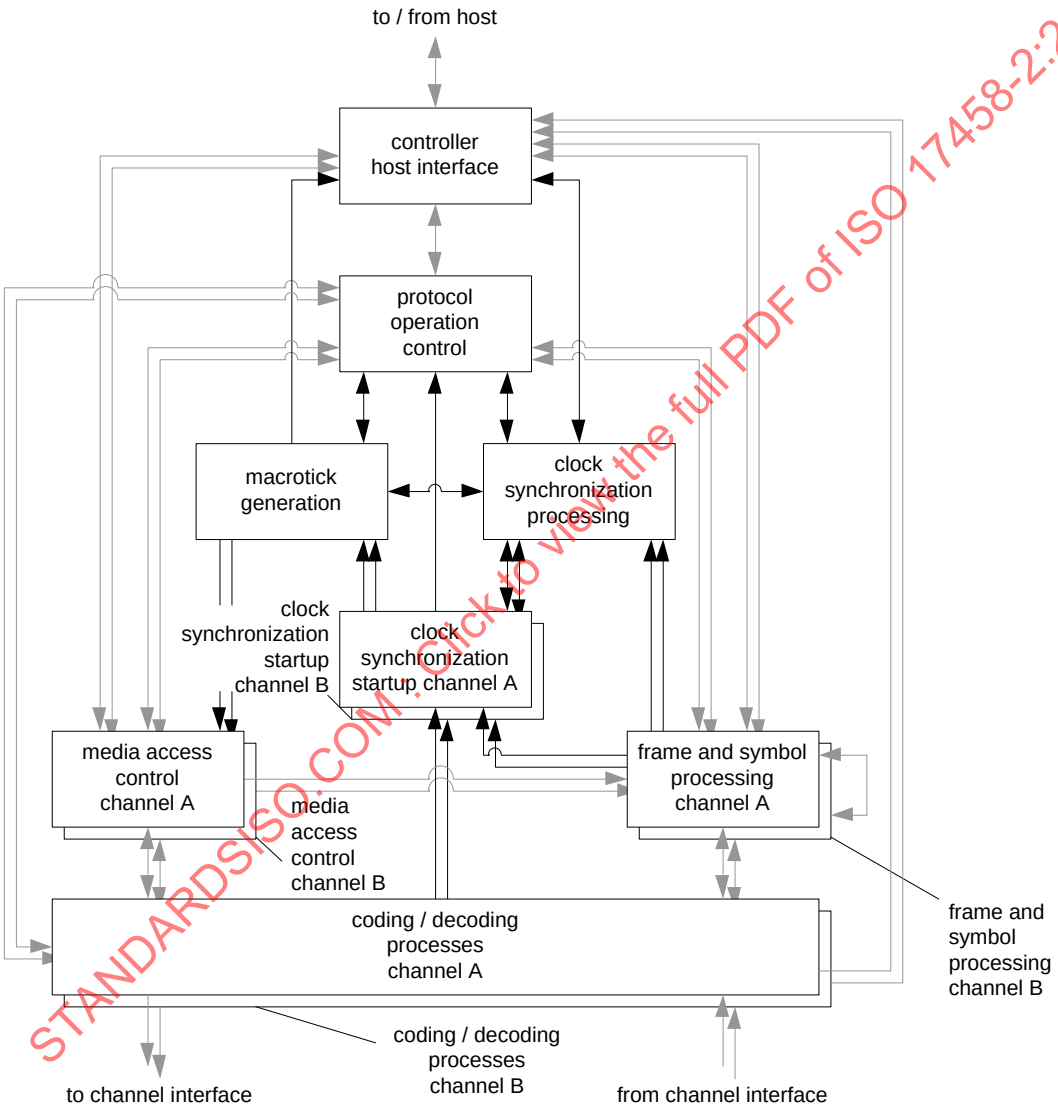





**Figure 152 — Clock synchronisation context**

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 synchronisation 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 synchronisation processes and the other protocol processes is depicted in Figure 152[105)].

## 12.2 Time representation

### 12.2.1 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 153).

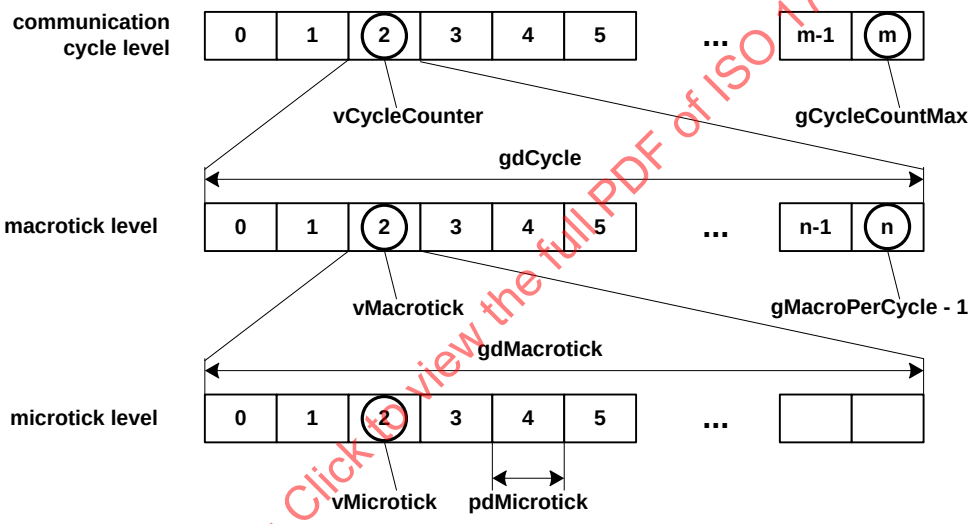



**Figure 153 — Timing hierarchy**

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)[106)].

105) The dark lines represent data flows between mechanisms that are relevant to this subclause. The lighter gray lines are relevant to the protocol, but not to this subclause.

106) This is true even for the nominal (uncorrected) average duration of a macrotick (for example 6 000 microticks distributed over 137 macroticks).

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 synchronisation in the cluster)[107].

### 12.2.2 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*[108].

The local time is based on the local view of the global time. Every node uses the clock synchronisation 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.

### 12.2.3 Parameters and variables

*vCycleCounter* is the (controller-local) cycle number and is incremented by one at the beginning of each communication cycle. *vCycleCounter* ranges from 0 to *gCycleCountMax*. When *gCycleCountMax* is reached, the cycle counter *vCycleCounter* shall be reset to zero in the next communication cycle instead of being incremented.

*vMacrotick* represents the current value of the (controller-local) macrotick and ranges from 0 to (*gMacroPerCycle* - 1). *gMacroPerCycle* defines the (integer) number of macroticks per cycle.

*vMicrotick* represents the current value of the (controller-local) microtick.

**Definition: *T_Macrotick* and *T_Microtick*** **(41)**

```
syntype
   T_Macrotick = Integer
endsyntype;

syntype
   T_Microtick = Integer
endsyntype;
```

The FlexRay "timing" will be configured by

— *gCycleCountMax*,

— *gMacroPerCycle*, and

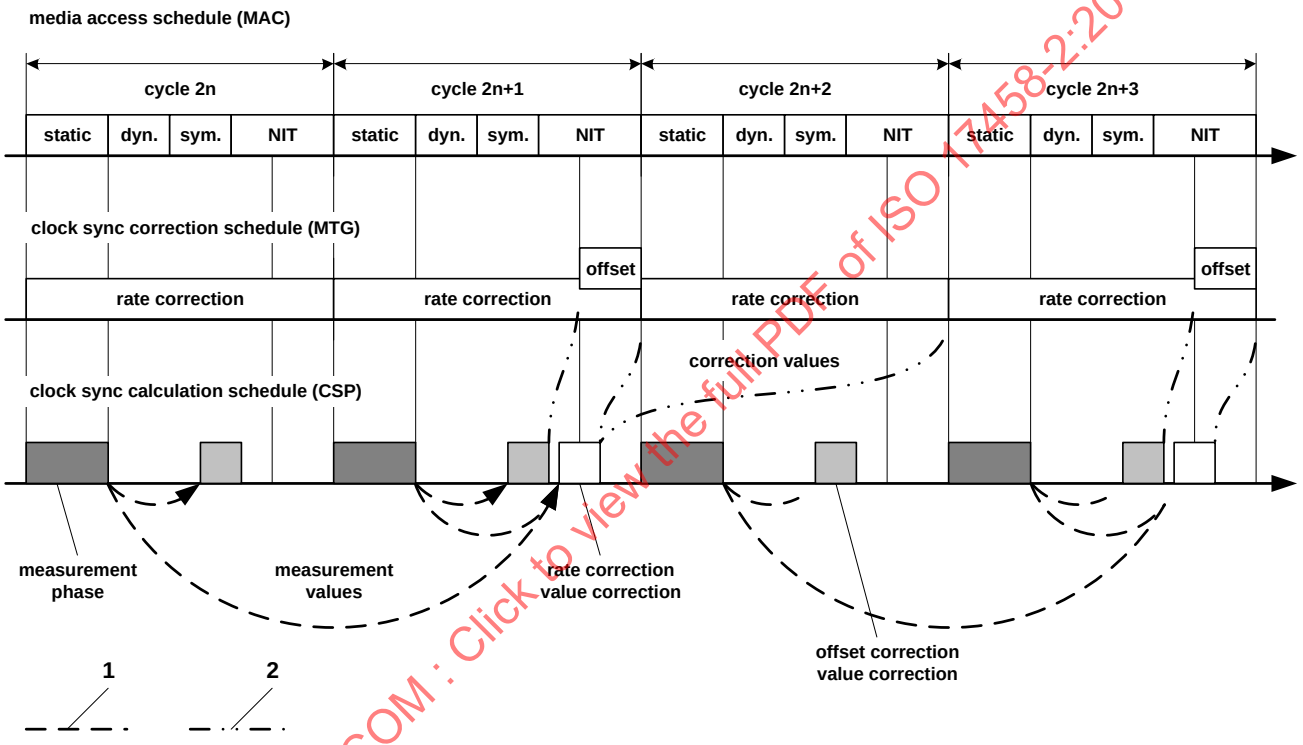
— two of the three parameters *pMicroPerCycle*, *gdCycle* and *pdMicrotick*. *pMicroPerCycle* is the node specific number of microticks per cycle, *gdCycle* is the cluster wide duration of one communication cycle, and pdMicrotick is the node specific duration of one microtick. The relation between these three parameters is described in B.4.16.

107) The cycle number discrepancy is at most one, and lasts no longer than the precision of the system.
108) An atomic action is an action where no interruption is possible.

## 12.3 Synchronisation process

Clock synchronisation 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 12.7. The clock synchronisation process (CSP) performs the initialisation at cycle start, the measurement and storage of deviation values, and the calculation of the offset and the rate correction values.

Figure 154 illustrates the timing relationship between these two processes and the relationship to the media access schedule.




**Key**

1 measurement values

2 correction values

**Figure 154 — Timing relationship between clock synchronisation operations and media access schedule**

The primary task of the clock synchronisation 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 between

— 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 shall be fulfilled.

— Rate correction and offset correction shall be done in the same way in all nodes. Rate correction shall be performed over the entire cycle.

— Offset correction shall be performed only during the NIT in the odd communication cycle, starts at *pOffsetCorrectionStart*, and shall be finished before the start of the next communication cycle.

— Offset changes (implemented by synchronizing the start time of the cycle) are described by the variable *zOffsetCorrection*. *zOffsetCorrection* indicates the number of microticks that are added to the offset correction segment of the network idle time. *zOffsetCorrection* may be negative. The value of *zOffsetCorrection* is determined by the clock synchronisation algorithm. The calculation of *zOffsetCorrection* takes place every cycle but a correction is only applied at the end of odd communication cycles. The calculation of *zOffsetCorrection* is based on values measured in a single communication cycle. Although the SDL indicates that this computation cannot begin before the NIT, an implementation may start the computation of this parameter within the dynamic segment or symbol window as long as the reaction to the computation (update of the CHI and transmission of the *SyncCalcResult* and *offset calc ready* signals) is delayed until the NIT. The calculation shall be complete before the offset correction phase begins.

— Rate (frequency) changes are described by the variable *zRateCorrection*. *zRateCorrection* is an integer number of microticks that are added to the configured number of microticks in a communication cycle (*pMicroPerCycle*)[109] *zRateCorrection* may be negative. The value of *zRateCorrection* is determined by the clock synchronisation algorithm and is only computed once per double cycle. The calculation of *zRateCorrection* takes place following the static segment in an odd cycle. The calculation of *zRateCorrection* is based on the values measured in an even-odd double cycle. Although the SDL indicates that this computation cannot begin before the NIT, an implementation may start the computation of this parameter within the dynamic segment or symbol window as long as the reaction to the computation (update of the CHI and transmission of the *SyncCalcResult* and *rate calc ready* signals) is delayed until the NIT. The calculation shall be completed before the next even cycle begins.

The following data types will be used in the definition of the clock synchronisation process:

**Definition: *T_EvenOdd* and *T_Deviation*** **(42)**

```
newtype T_EvenOdd
   literals even, odd;
endnewtype;

syntype
   T_Deviation = T_Microtick
endsyntype;
```

The protocol operation control (POC) process sets the operating mode for the clock synchronisation process (CSP) (Figure 156) into one of the following modes.

— In the STANDBY mode the clock synchronisation process is effectively halted.

— In the NOSYNC mode CSP performs clock synchronisation under the assumption that it is not transmitting sync frames (i.e., it does not include its own clock in the clock correction computations).

— In the SYNC mode CSP performs clock synchronisation under the assumption that it is transmitting sync frames (i.e., it includes its own clock in the clock correction computations).

109) *pMicroPerCycle* is the configured number of microticks per communication cycle without correction.

Definition **(43)** gives the formal definition of the CSP operating modes.

**Definition: *T_CspMode* and *T_SyncCalcResult*** **(43)**

```
newtype T_CspMode
   literals STANDBY, NOSYNC, SYNC;
endnewtype;

newtype T_SyncCalcResult
   literals WITHIN_BOUNDS, EXCEEDS_BOUNDS, MISSING_TERM;
endnewtype;
```

After the POC sets the CSP mode to something other than STANDBY, the CSP waits in the *CSP:wait for startup* state until the POC forces the node to a coldstart or to integrate into a cluster. The startup procedure, including its initialisation and interaction with other processes, is described in the macro INTEGRATION_CONTROL, which is explained in 12.4.

Before further explanation of the processes an array is defined (Definition **(44)**), which is used to store the frame IDs of the sync frames that are considered in the clock correction process.

**Definition: *T_ArrayIndex, T_SyncFrameIDCount*, and *T_FrameIDTable*** **(44)**

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

syntype T_SyncFrameIDCount = Integer
   constants 0 : cSyncFrameIDCountMax
endsyntype;

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

**Definition: *T_SyncExtern*** **(45)**

```
newtype T_SyncExtern
   literals UNSYNC, ACTIVE, PASSIVE;
endnewtype;
```

Figure 155 depicts the declarations for the clock synchronisation process [CSP].

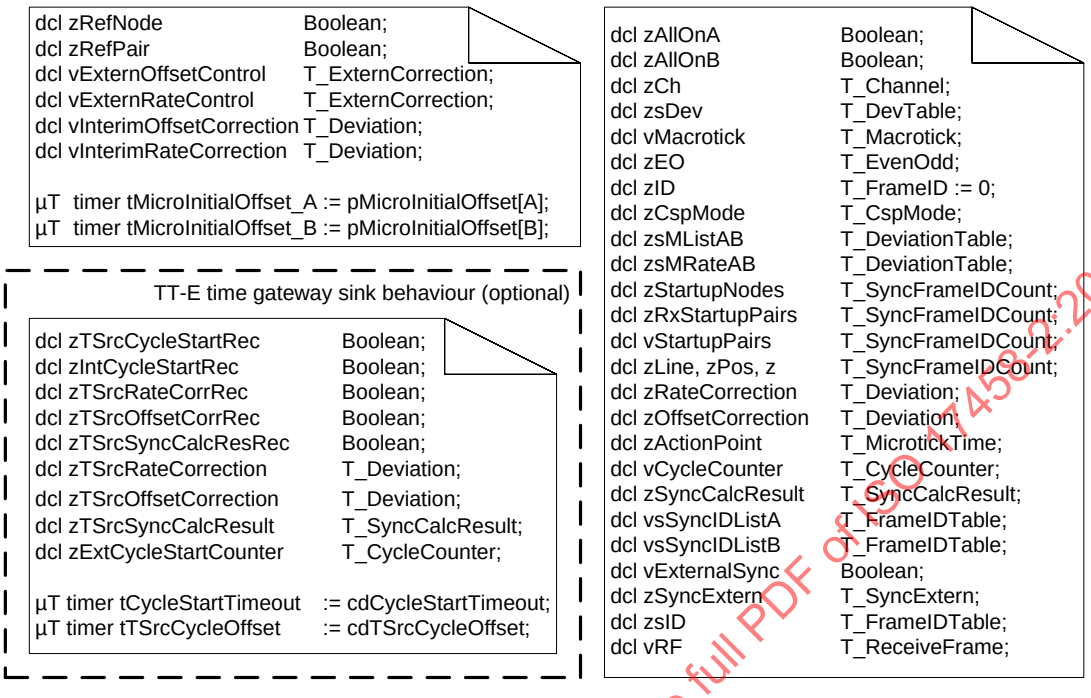



**Figure 155 — Declarations for the clock synchronisation process [CSP]**

Figure 156 depicts the start of the clock synchronisation process [CSP].

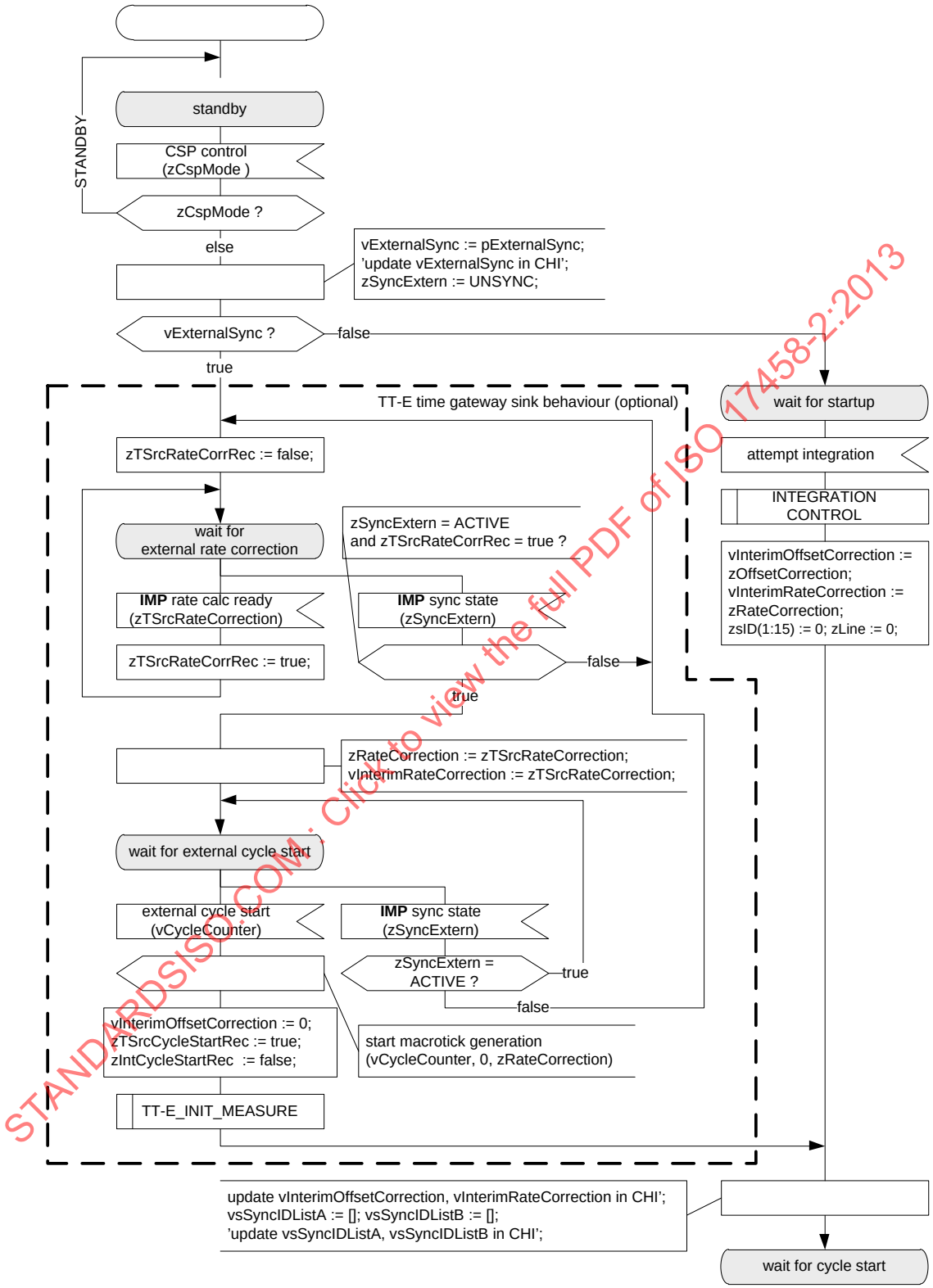



**Figure 156 — Start of the clock synchronisation process [CSP]**

Figure 157 depicts the wait for cycle start [CSP].

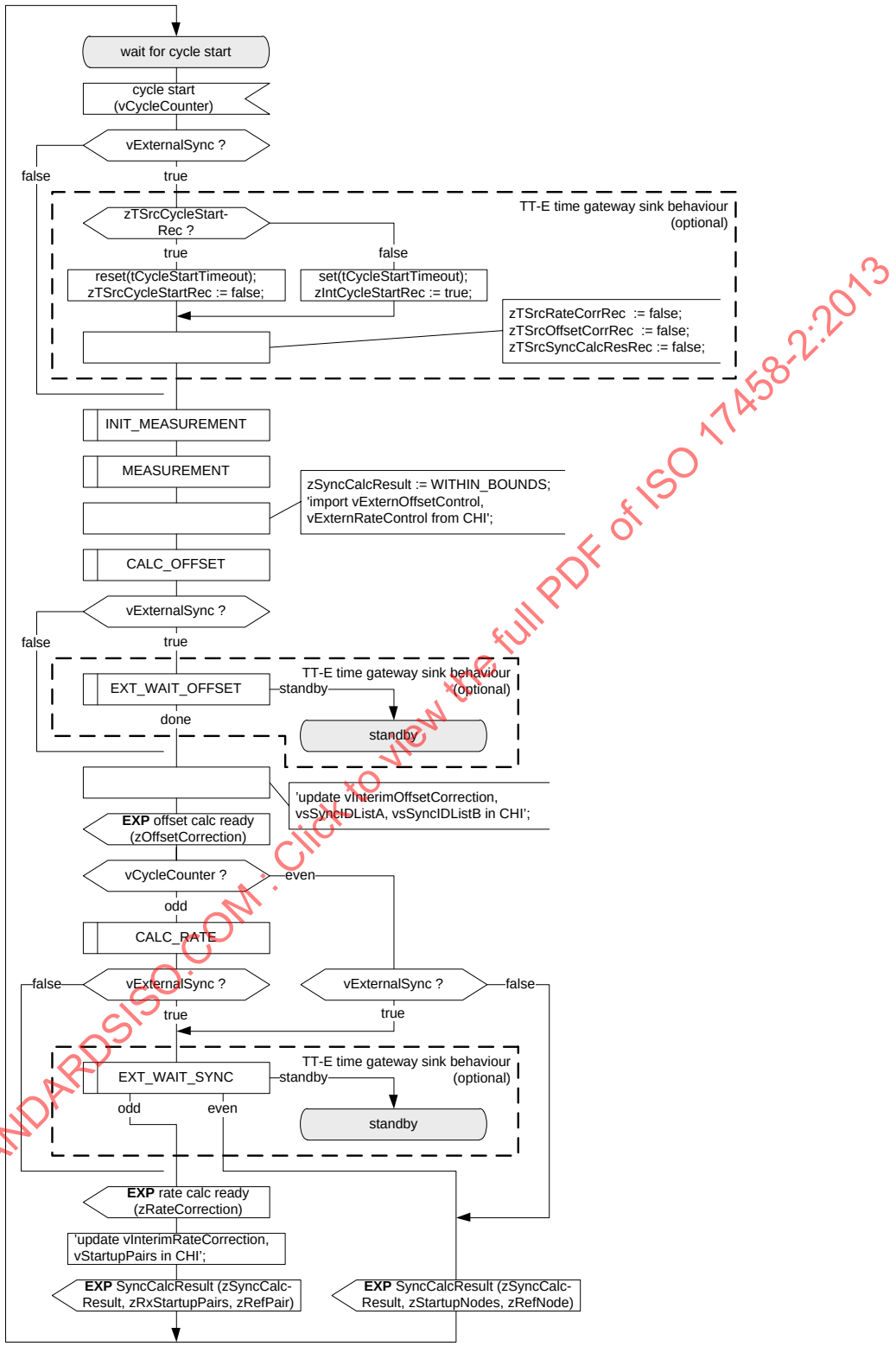



**Figure 157 — Wait for cycle start [CSP]**

Figure 158 depicts the measurement initialization for TT-E coldstart [CSP].

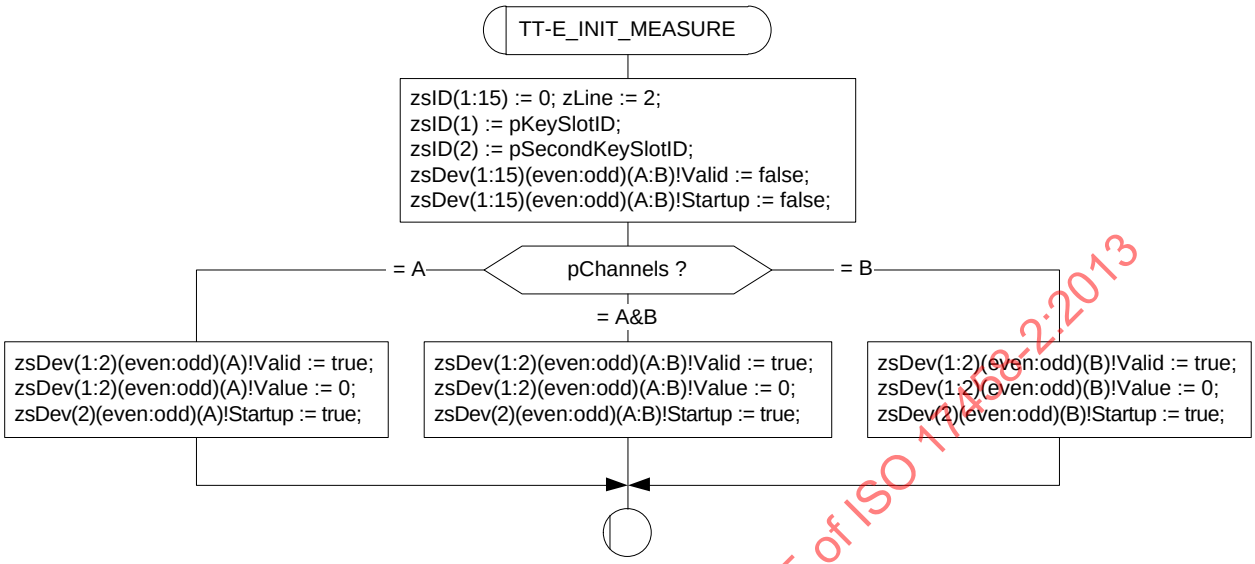



**Figure 158 — Measurement initialization for TT-E coldstart [CSP]**

After finishing the startup procedure a repetitive sequence consisting of cycle initialisation (see Figure 168), a measurement phase (see Figure 165), and offset (see Figure 144) and rate (see Figure 169) 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 synchronisation control (see Figure 159) handles mode changes done by the POC. It also handles process termination requests sent by the POC.

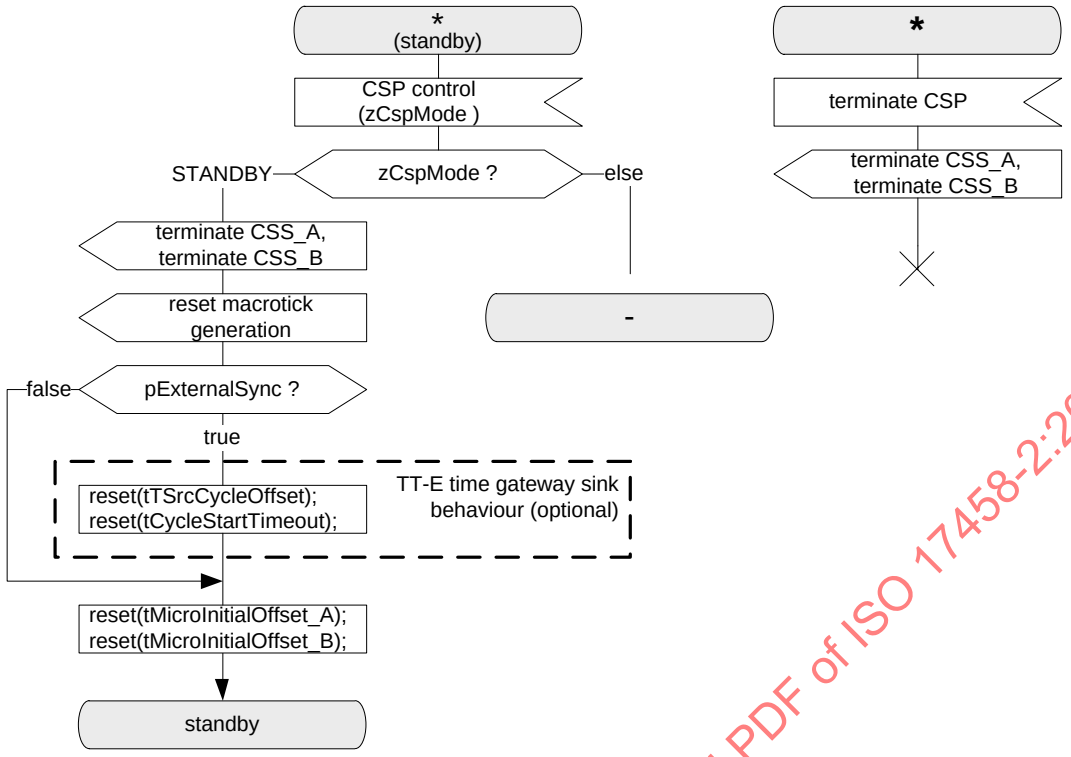



**Figure 159 — Clock synchronisation control and termination [CSP]**

## 12.4 Startup of the clock synchronisation

### 12.4.1 Preconditions and startup types

The startup of the node's clock synchronisation requires

— the initialisation and start of the MTG process and

— the initialisation and start of the CSP process. This process contains the repetitive tasks of measurement and storage of deviation values and the calculation of the offset and the rate correction values.

There are two ways to start the clock synchronisation of a node.

— The node is the leading coldstart node.

— The node adopts the initialisation values (cycle counter, clock rate, and cycle start time) of a running coldstart node.

Figure 160 depicts the integration control [CSP].

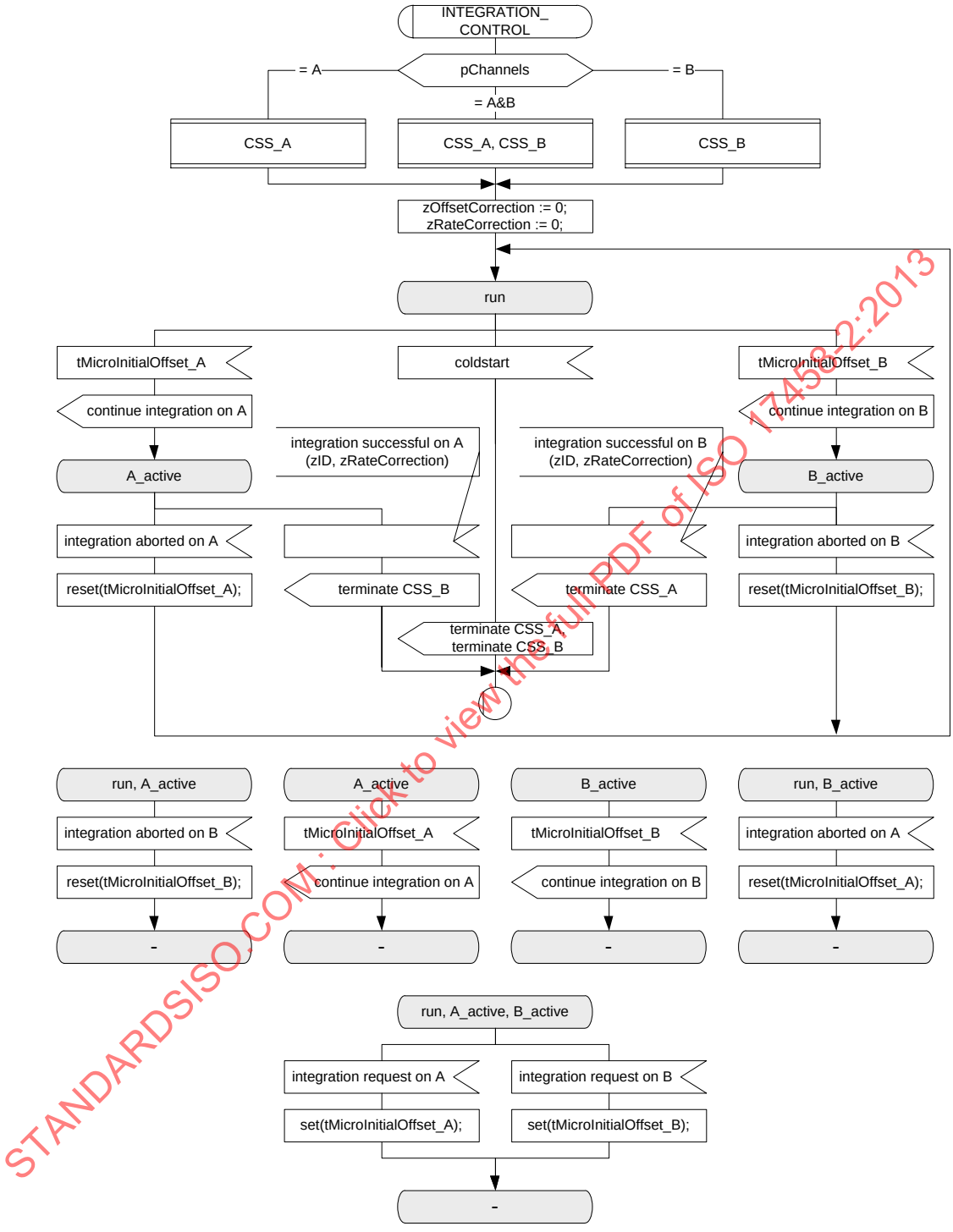





**Figure 160 — Integration control [CSP]**

The startup procedure will be entered when the CSP receives the signal *attempt integration* from the POC (see Figure 156). The control of the node's startup is described in the INTEGRATION_CONTROL macro depicted in Figure 160.

### 12.4.2 Coldstart startup

If ongoing communication on the channels is not detected the POC may force the node to perform the role of the leading coldstart node of the cluster. This causes the following actions.

— The clock synchronisation 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) (see Figure 176) leaves the *MTG:wait for start* state. Depending on the initialisation 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.

### 12.4.3 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 (see Figure 160) instantiates the clock synchronisation startup processes for channel A and B (CSS_A, CSS_B).

After their instantiation, the CSS_A process (see Figure 162) 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 (see Figure 176) 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)[110)]. 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 behaviour of the CSS process of the non-initial channel is the same as the behaviour 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.

Figure 161 depicts the termination of the CSS process [CSS_A].

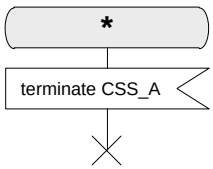



**Figure 161 — Termination of the CSS process [CSS_A]**

110) 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 (see Figure 162).

Figure 162 depicts the clock synchronisation startup process on channel A [CSS_A].

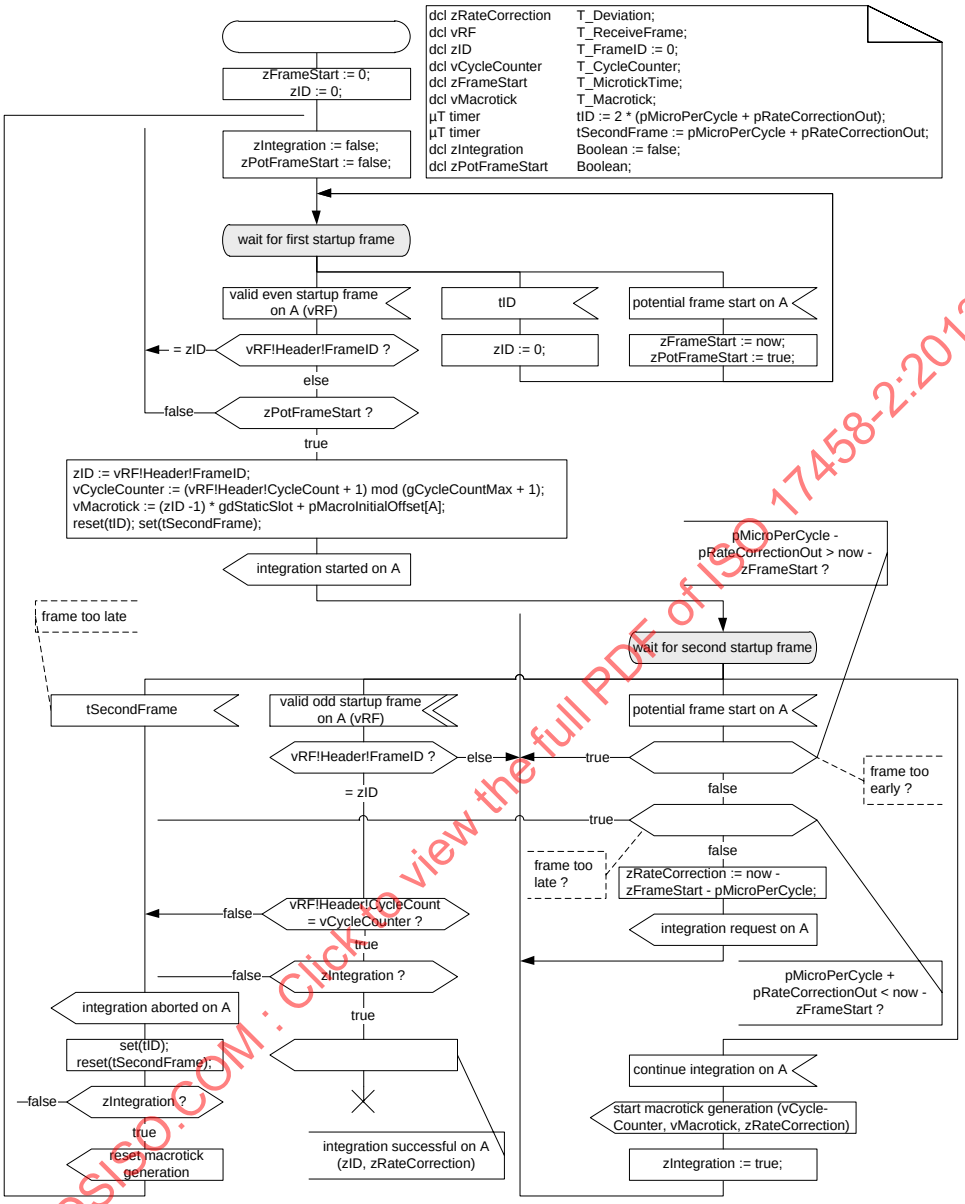



**Figure 162 — Clock synchronisation startup process on channel A [CSS_A][111)]**



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 160). This behaviour of this termination is depicted in Figure 161.

The timer *tSecondFrame* in Figure 162 is used to restart the clock synchronisation startup process if the corresponding odd startup frame was not received after an appropriate period of time.

111) The priority input symbol on the *CSS_A:wait for second startup frame* state has been included to resolve the ambiguity that arises if the timer *tSecondFrame* expires at the same time a *valid odd startup frame on A* signal is received.

The 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.

## 12.5 Time measurement

### 12.5.1 General

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 12.5.2. This data structure is used in the explanation of the initialisation (12.5.3) and the measurement, storage, and deviation calculation mechanisms (12.5.4).

### 12.5.2 Data structure

The following data types are introduced to enable a compact description of mechanisms related to clock synchronisation:

**Definition: *T_DevValid*** **(46)**

```
newtype T_DevValid
struct
   Value       T_Deviation;
   Valid       Boolean;
   Startup     Boolean;
endnewtype;
```

**Definition: *T_ChannelDev*, *T_EOChDev*, and *T_DevTable*** **(47)**

```
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;
```

The structured data type *T_DevTable* is a three dimensional array with the dimensions line number (1 …15), communication channel (A or B), and communication cycle (even or odd). Each line is used to store the received data of one sync frame pair transmitted by the same node in the same slot in subsequent cycles.

If the node is itself a sync node the first line is used to store a deviation of zero, corresponding to the deviation of its own sync frame. TT-E and TT-L sync nodes store zero deviations in the first two lines of the table, one for each of the two sync frames transmitted.

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 163 gives an example of this data structure.



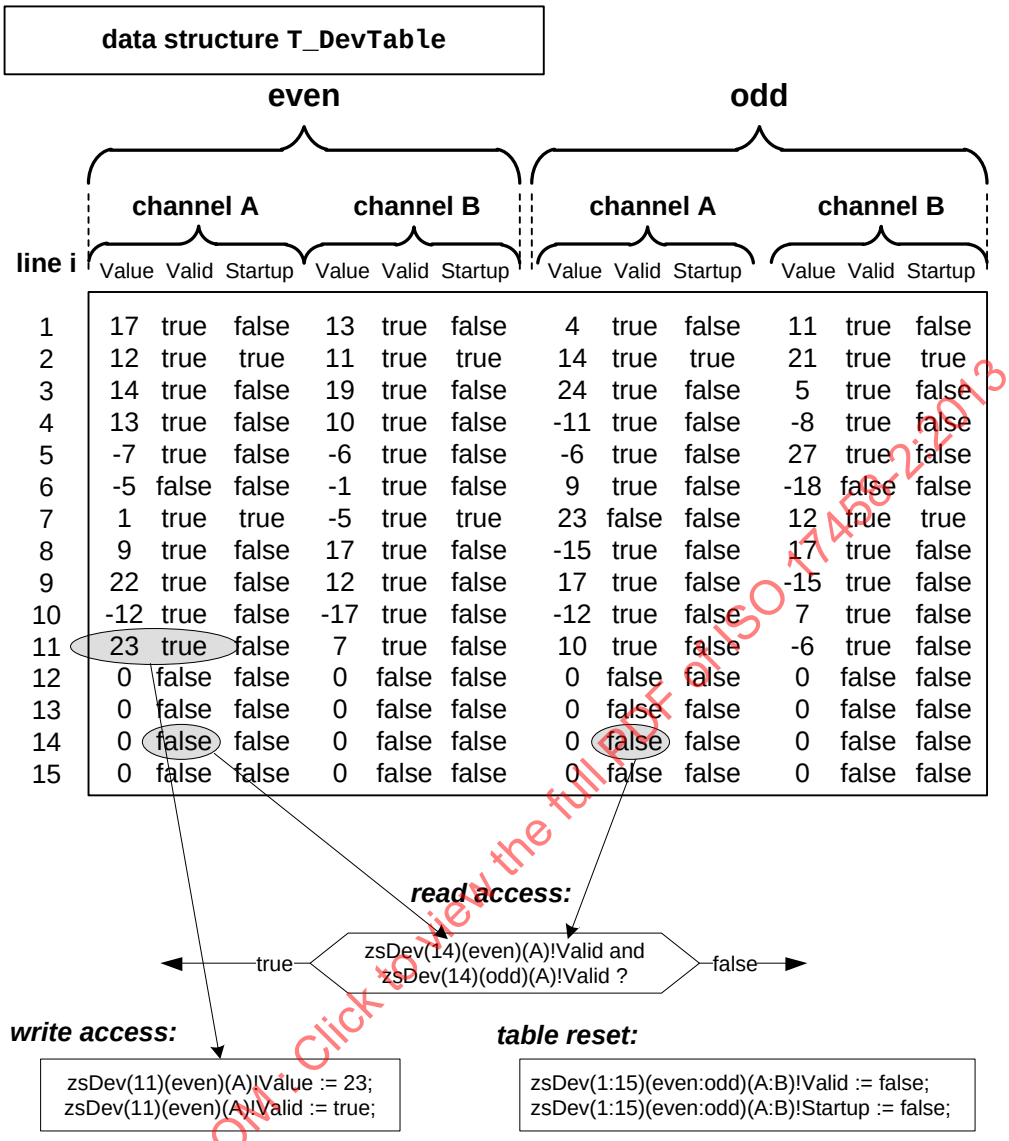



**Figure 163 — Data structure example**



### 12.5.3 Initialisation

The data structure introduced in 12.5.2 is used to instantiate a variable (*zsDev*). The variable *zsDev* will be initialized at the beginning of an even communication cycle[112)]. Additionally, if the node is configured to transmit sync frames (mode SYNC), corresponding entries are stored in the variable as depicted in Figure 164. If the node is operating in the two key slot mode (i.e., is either a TT-L coldstart node or a TT-E coldstart node), entries corresponding to the second transmitted startup frame are also added to the variable *zsDev*.

In contrast to the entries for the first sync frame, the entries for the second key slot are also flagged as belonging to a startup frame, which allows a TT-L coldstart node to proceed through the startup without having received any startup frames from other nodes (see clause 11).

112) TT-E coldstart nodes also initialize zsDev on the first cycle of operation regardless of whether it is an even or odd cycle. See Figure 156 and Figure 158 for details.

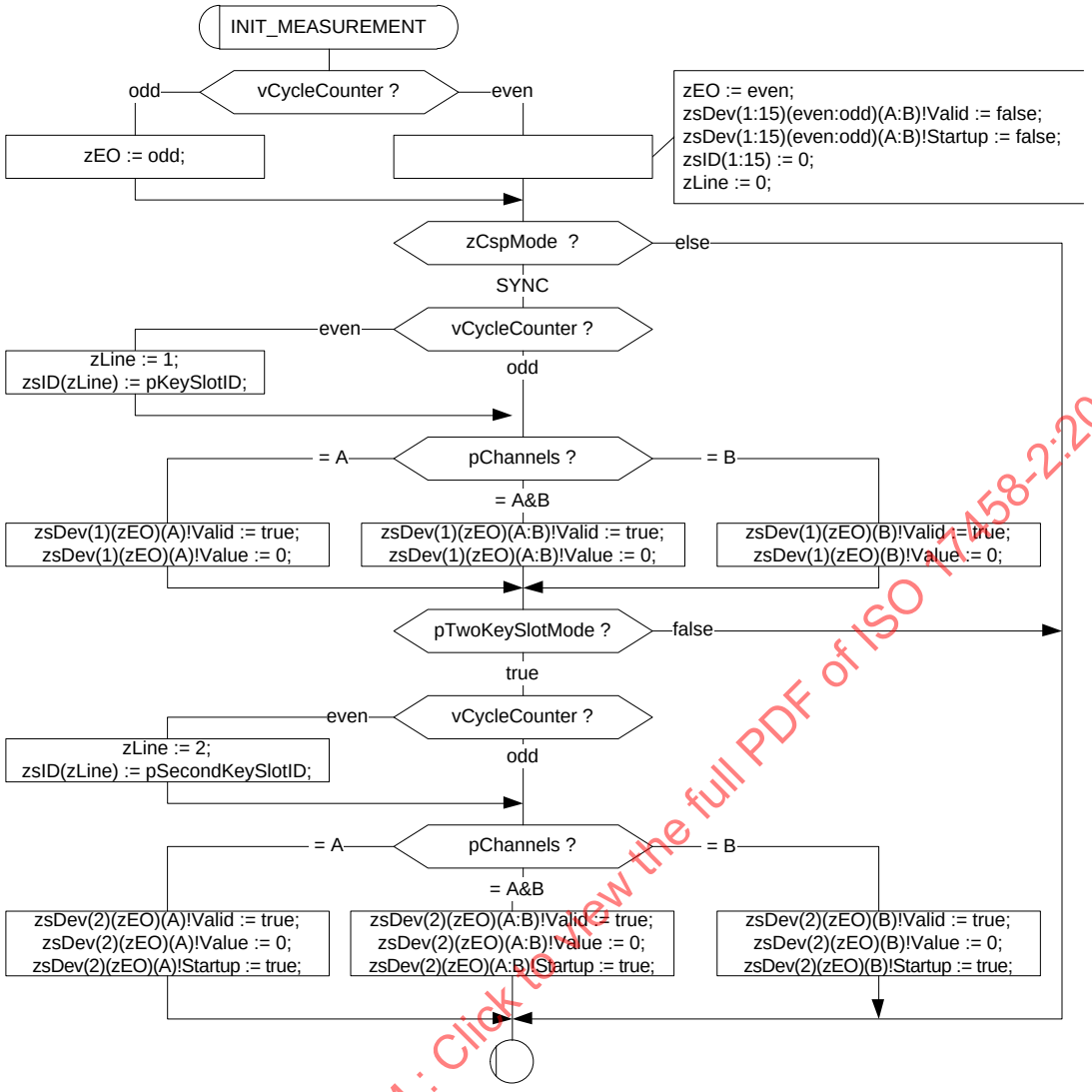



**Figure 164 — Initialisation of the data structure for measurement [CSP]**

### 12.5.4 Time measurement storage

The expected arrival time of a static frame is the static slot action point, which is defined in clause 9. The MAC generates a signal when the static slot action point is reached. When the clock synchronisation 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 7.2.6.

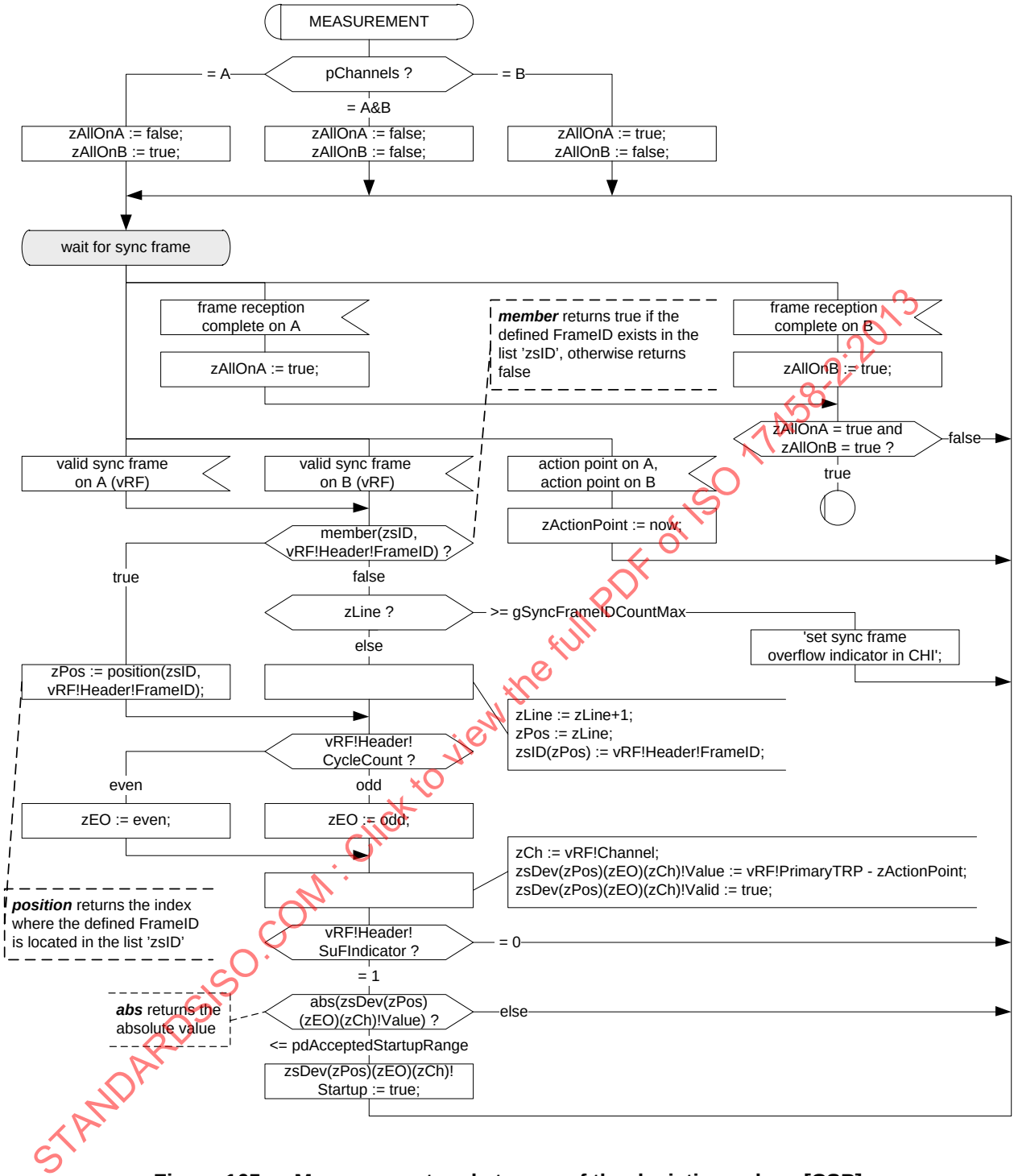





**Figure 165 — Measurement and storage of the deviation values [CSP]**

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 165). The measurement phase ends when the static segment ends.

The reception[113)] of more than *gSyncFrameIDCountMax* unique sync frame identifiers in either a single communication cycle or an even / odd communication cycle pair indicates an error inside the cluster. This is reported to the host and only measurements corresponding to the first *gSyncFrameIDCountMax* unique sync frame identifiers are considered for the correction value calculations.

## 12.6 Correction term calculation

### 12.6.1 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 166 and Figure 167):

a) The algorithm determines the value of a parameter, k, based on the number of values in the sorted list (see Table 5)[114)].

**Table 5 — FTM term deletion as a function of list size**

| Number of values | k |
|---|---|
| 1 - 2 | 0 |
| 3 – 7 | 1 |
| > 7 | 2 |

b) The measured values are sorted and the k largest and the k smallest values are discarded.

c) The largest and the smallest of the remaining values are averaged for the calculation of the midpoint value. Note that the division by two of odd numbers should truncate towards zero such that the result is an integer number[115)]. The resulting value is assumed to represent the node's deviation from the global time base and serves as the correction term.

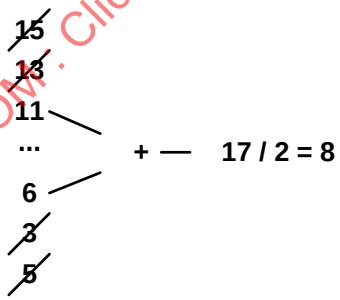



**Figure 166 — Example clock correction value calculation with k=2**



113) Nodes that transmit sync frames, and are operating in the *POC:normal active* state, will also use their own sync frame measurement values in clock correction. For the purposes of this check, the node's own sync frame is considered to be implicitly received, even though there is no actual reception. As a result, the node's own sync frame is included in the count of sync frame identifiers that is checked against *gSyncFrameIDCountMax*.

114) The parameter k is not the number of asymmetric faults that can be tolerated.

115) Example: 17 / 2 = 8 and -17 / 2 = -8.

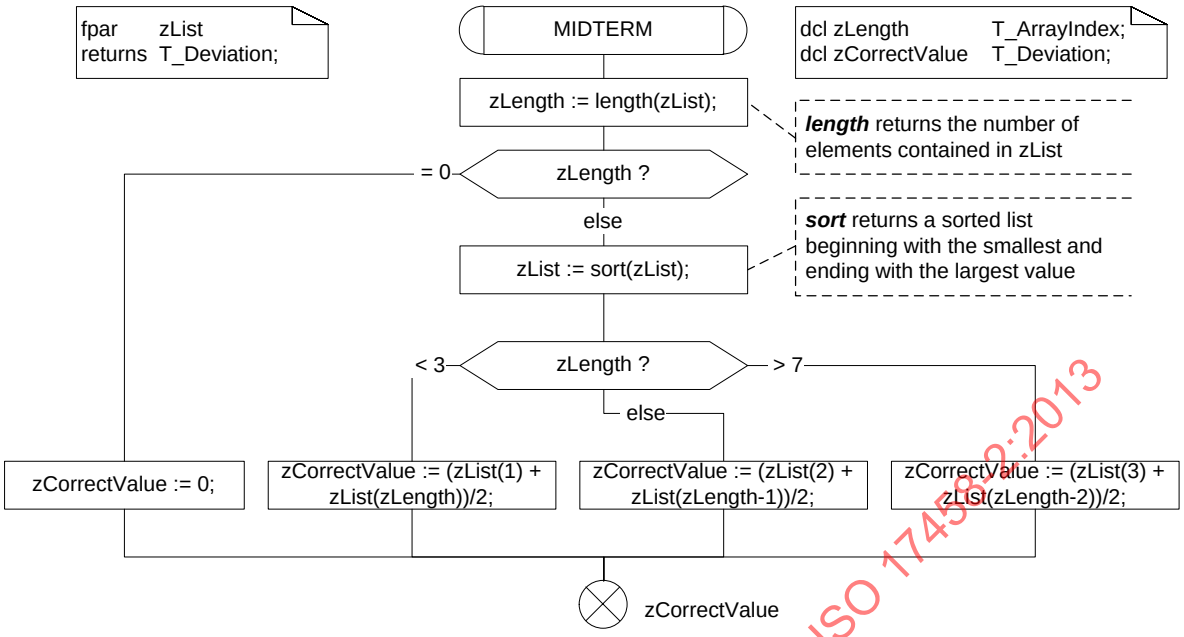



**Figure 167 — Fault-Tolerant Midpoint Procedure [CSP]**

### 12.6.2 Calculation of the offset correction value

The offset correction value *zOffsetCorrection* is a signed integer that indicates how many microticks the node should shift the start of its cycle. Negative values mean the NIT should be shortened (making the next cycle start earlier). Positive values mean the NIT should be lengthened (making the next cycle start later).

In Figure 168 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 168.

a) 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.

b) The number of received sync frames is checked and if an insufficient[116)] number of sync frames was received the error condition MISSING_TERM is set.

c) The fault-tolerant midpoint algorithm is executed (see 12.6.1).

d) The correction term is checked against specified limits. If the correction term is outside of the specified limits the error condition is set to EXCEEDS_BOUNDS and the correction term is set to the maximum or minimum value as appropriate (see 12.6.4).

e) If appropriate, an external correction value supplied by the host is added to the calculated and checked correction term (see 12.6.5).

116) The number of sync frames that need to be received to be considered sufficient depends on the synchronisation mode and the node's role. For devices in a TT-D cluster, and for non-coldstart nodes in a TT-E or TT-L cluster, the reception of a single valid sync frame is considered sufficient. TT-E or TT-L coldstart nodes do not need to receive any sync frames, i.e., the number of received sync frames is always considered sufficient. Figure 168 implements this behaviour not by checking the number of received sync frames directly but rather by checking the number of entries in the *zsMListAB* array.

The following data structure is used to save and handle the selected data:

**Definition: *T_DeviationTable*** **(48)**

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

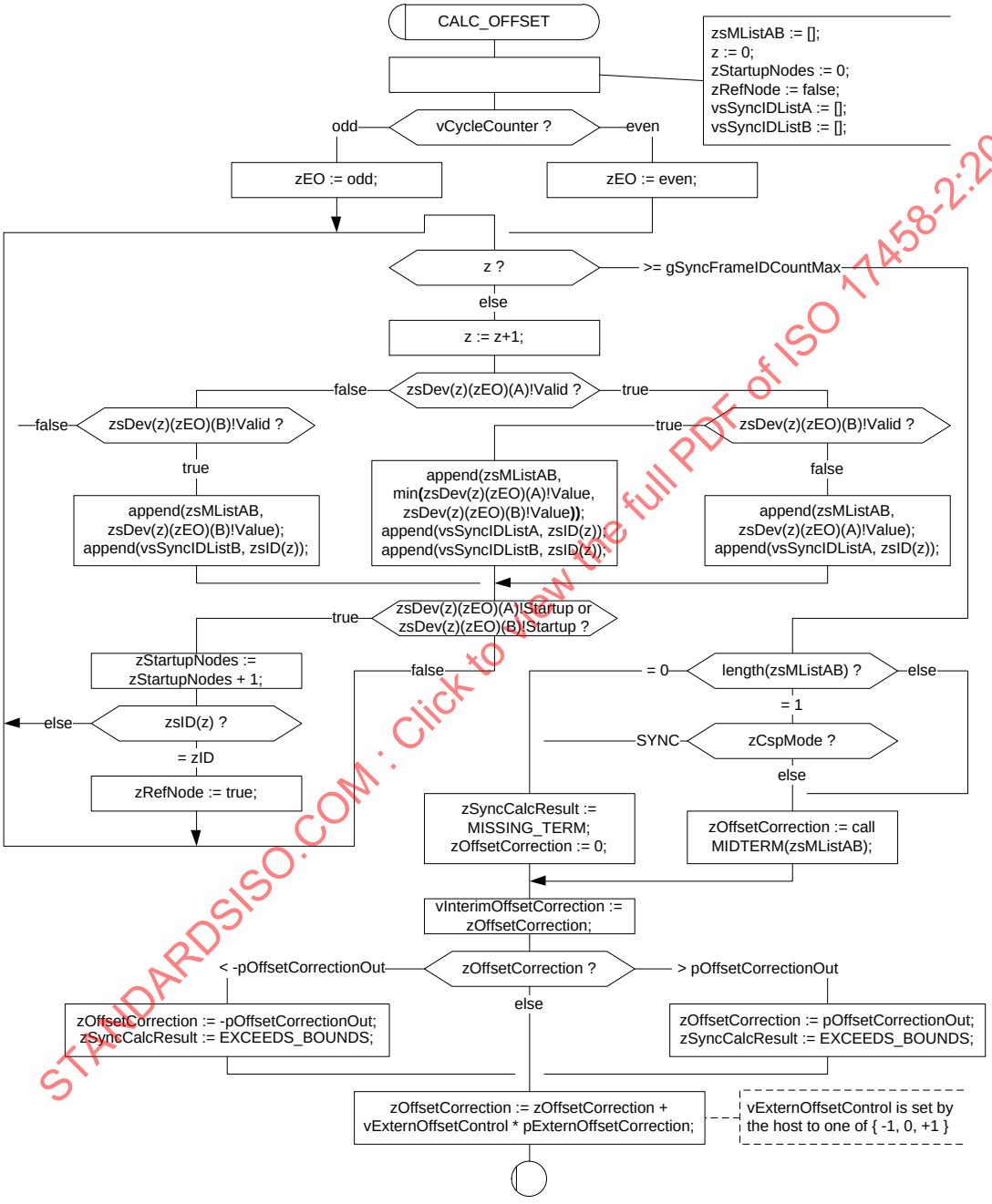





**Figure 168 — Calculation of the offset correction value [CSP]**

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 157 (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 shall 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.

### 12.6.3 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 169.

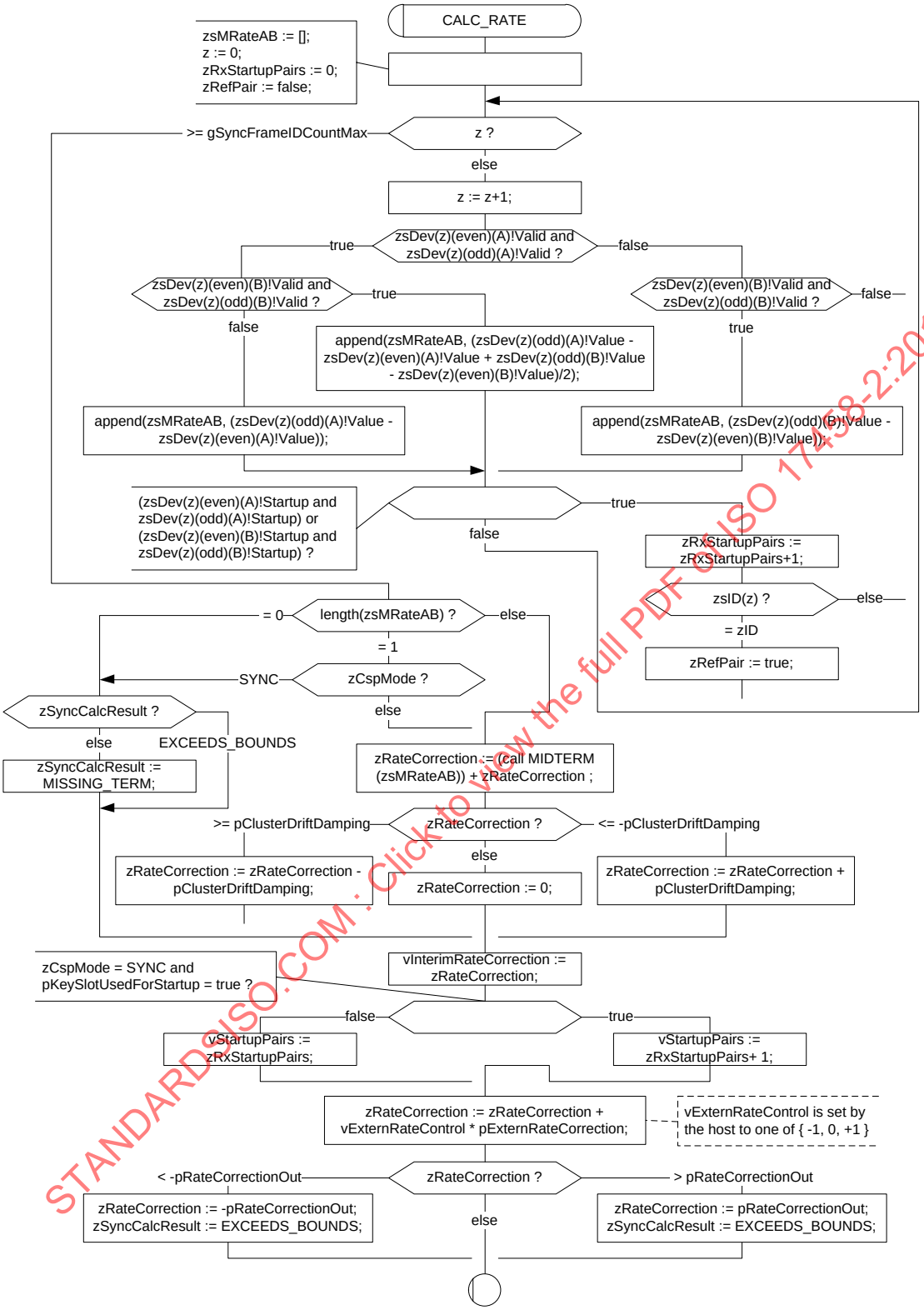





**Figure 169 — Calculation of the rate correction value [CSP]**

The rate correction value *zRateCorrection* is a signed integer indicating by how many microticks the node's cycle length should be changed. Negative values mean the cycle should be shortened; positive values mean the cycle should be lengthened.

The following steps are depicted in the SDL diagram in Figure 169.

a) Pairs of previously stored deviation values are selected and the difference between the values within a pair is calculated. Pairs are selected that represent sync frames received on the same channel, in slots with the same slot number, on consecutive cycles. If there are two pairs for a given sync frame ID (one pair for channel A and another pair for channel B) the average of the differences is used.

b) The number of received sync frame pairs is checked and if an insufficient[117] number of sync frame pairs was received the error condition MISSING_TERM is set unless the error condition EXCEEDS_BOUNDS had been set previously[118].

c) The fault-tolerant midpoint algorithm is executed (see 12.6.1).

d) A damping value *pClusterDriftDamping* for the rate correction term is applied.

e) If appropriate, an external correction value supplied by the host is added to the calculated correction term (see 12.6.5).

f) The correction term is checked against specified limits. If the correction term exceeds the specified limits the error condition is set to EXCEEDS_BOUNDS and the correction term is set to the maximum or minimum value as appropriate (see 12.6.4).

In the list above division operations shall produce an integral result that is truncated towards zero[119].

The *pClusterDriftDamping* parameter should be configured in such a way that the damping value in all nodes has approximately the same duration[120].

The SDL abstraction of execution in zero time introduces similar problems for the rate correction calculation as are described in 12.6.2 for the offset correction calculation. Therefore the following restriction on the time required for rate correction calculation is introduced:

The rate correction calculation shall 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.

### 12.6.4 Value limitations

Before applying the calculated correction values, they shall be checked against pre-configured limits (see Figure 168 and Figure 169).

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 synchronisation. This corresponds to an error condition. Information on the handling of this situation is specified in clause 6.



117) The number of sync frame pairs that need to be received to be considered sufficient depends on the synchronisation mode and the node's role. For devices in a TT-D cluster, and for non-coldstart nodes in a TT-E or TT-L cluster, the reception of a single valid sync frame pair is considered sufficient. TT-E or TT-L coldstart nodes do not need to receive any sync frames, i.e., the number of received sync frame pairs is always considered sufficient. Figure 169 implements this behaviour not by checking the number of received sync frame pairs directly but rather by checking the number of entries in the *zsMRateAB* array.

118) The consequence of this behaviour is that in a TT-D cluster operating with only a single sync node the sync node would detect MISSING_TERM before the non-sync nodes. The non-sync nodes would not detect MISSING_TERM until the last sync node has stopped transmitting, presumably due to loss of synchronisation. As a consequence, in this circumstance different types of nodes would cease operation at different times.

119) Example: 17 / 2 = 8 and -17 / 2 = -8.

120) A node-specific configuration value is used to allow clusters that have different microtick durations in different nodes.

The correction values are inside the limits if:

- pRateCorrectionOut <= zRateCorrection <= pRateCorrectionOut

- pOffsetCorrectionOut <= zOffsetCorrection <= pOffsetCorrectionOut

If both correction values are inside the limits the correction will be performed; if either value exceeds its preconfigured limit, an error is reported and the node enters the *POC:normal passive* or the *POC:halt* state depending on the configured behaviour (see clause 6). 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.

### 12.6.5 Host-controlled external clock synchronisation

During normal operation independent clusters can drift significantly. If synchronous operation is desired across multiple clusters, external synchronisation is necessary even though the nodes within each cluster are synchronized. This can be accomplished by the synchronous application of host-controlled external rate and offset correction terms to both clusters.

The control data *vExternRateControl* and *vExternOffsetControl* of the external clock correction have three different values, +1 / -1 / 0 with the following meanings as specified in Table 6.

**Table 6 — External clock correction control**

| Value | +1 | -1 | 0 |
|---|---|---|---|
| rate correction<br>*vExternRateControl* | increase cycle length by<br>*pExternRateCorrection* | decrease cycle length by<br>*pExternRateCorrection* | no change |
| offset correction<br>*vExternOffsetControl* | start cycle later by<br>*pExternOffsetCorrection* | start cycle earlier by<br>*pExternOffsetCorrection* | no change |

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 shall 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:

**Definition: *T_ExternCorrection*** **(49)**

```
syntype T_ExternCorrection = Integer
   constants -1, 0, +1
endsyntype;
```

The handling of the external correction values is depicted in Figure 168 and Figure 169.

The configuration shall 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.

### 12.6.6 TT-E time gateway sink correction determination

A node operating as TT-E coldstart node supports additional functionality which is described in Figure 170 to Figure 175. This functionality is optional and is shown in the SDL diagrams as a dashed outline box labelled "TT-E time gateway sink behaviour".

A TT-E coldstart node calculates the clock correction values as any other node in the cluster based on the measured deviation between the expected and the observed arrival times of the sync frames, even though the calculated clock correction values will eventually be overwritten by the clock correction values provided by the time gateway source node.

If a TT-E coldstart node is configured to switch to the local synchronisation method in case the synchronisation with the time gateway source node is lost, it will use the clock correction values it has calculated itself.

However, as a TT-E coldstart node configured to switch to the local synchronisation method shall necessarily be the sole coldstart node of the TT-E cluster, the calculated correction values will always be zero, just as they are for a TT-L coldstart node.

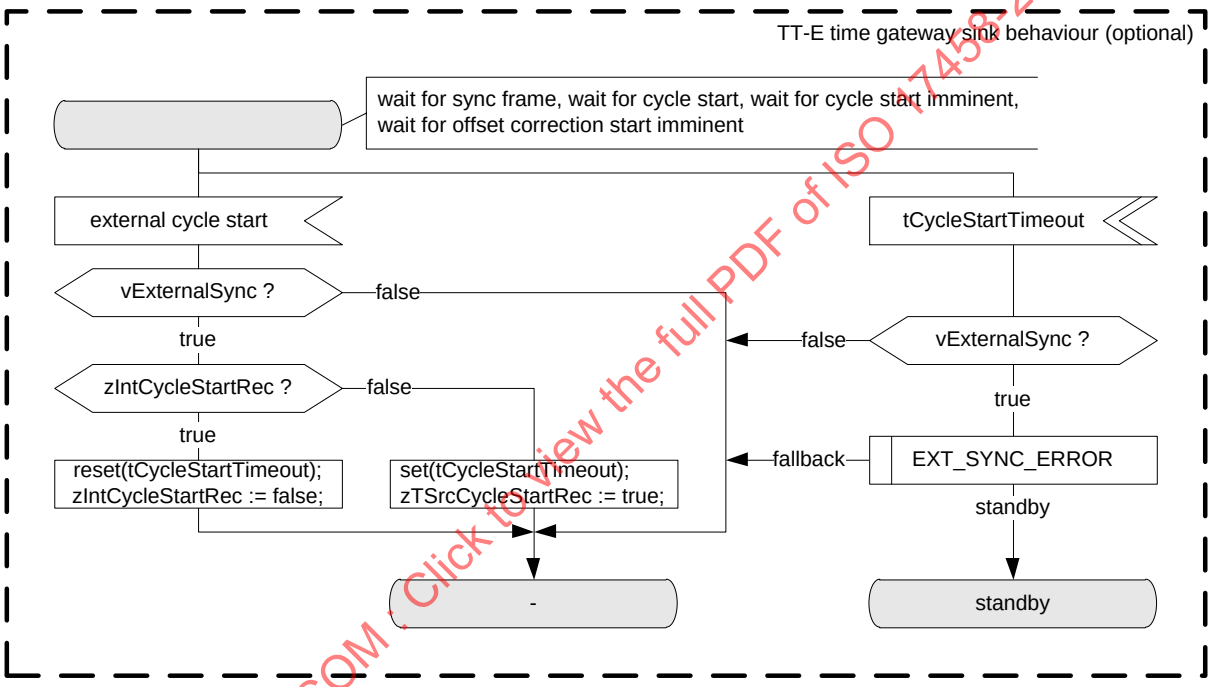



**Figure 170 — Cycle start supervision [CSP]**

Figure 170 above describes how the time gateway sink node continuously monitors that its locally generated *cycle start* signal is still synchronous to the one periodically sent by its time gateway source node.

Should the externally generated cycle start event and the locally generated cycle start event be *cdCycleStartTimeout* or more microticks apart, the time gateway sink node assumes that it has lost synchronisation to the time gateway source node (see Figure 175).

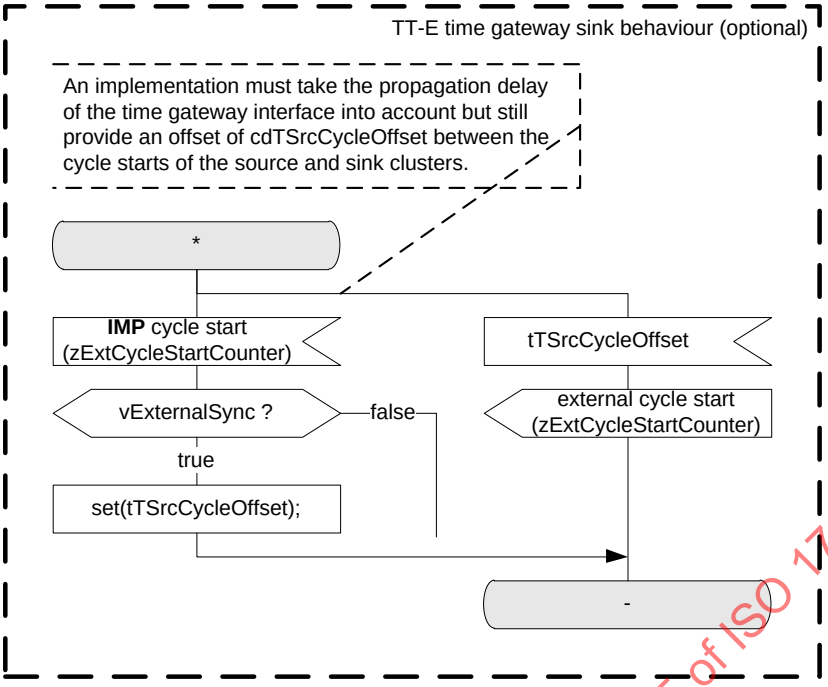



**Figure 171 — External cycle start delay [CSP]**

The delay timer *tTSrcCycleOffset* provides a delay between the cycle starts of the time gateway source and time gateway sink clusters. It is intended that this offset is a protocol constant (i.e., *cdTSrcCycleOffset*).

The SDL description makes the assumption that the cycle start event propagates across the time gateway interface in zero time, and thus the SDL describes a delay that is equal to the delay that is required between the two cycle starts. In practice, propagation across the time gateway interface will not take place in zero time – an implementation shall take this propagation time into account but still provide an offset of *cdTSrcCycleOffset* between the cycle starts in the two clusters.

This may require an implementation to provide one or more configuration parameters, not described in this specification, if the propagation delay across the time gateway interface is not under the control of the implementation (for example, if the time gateway interface is external to the implementation).

The time gateway sink node continuously monitors the time gateway interface and stores any received updates of clock correction values as depicted in Figure 172.

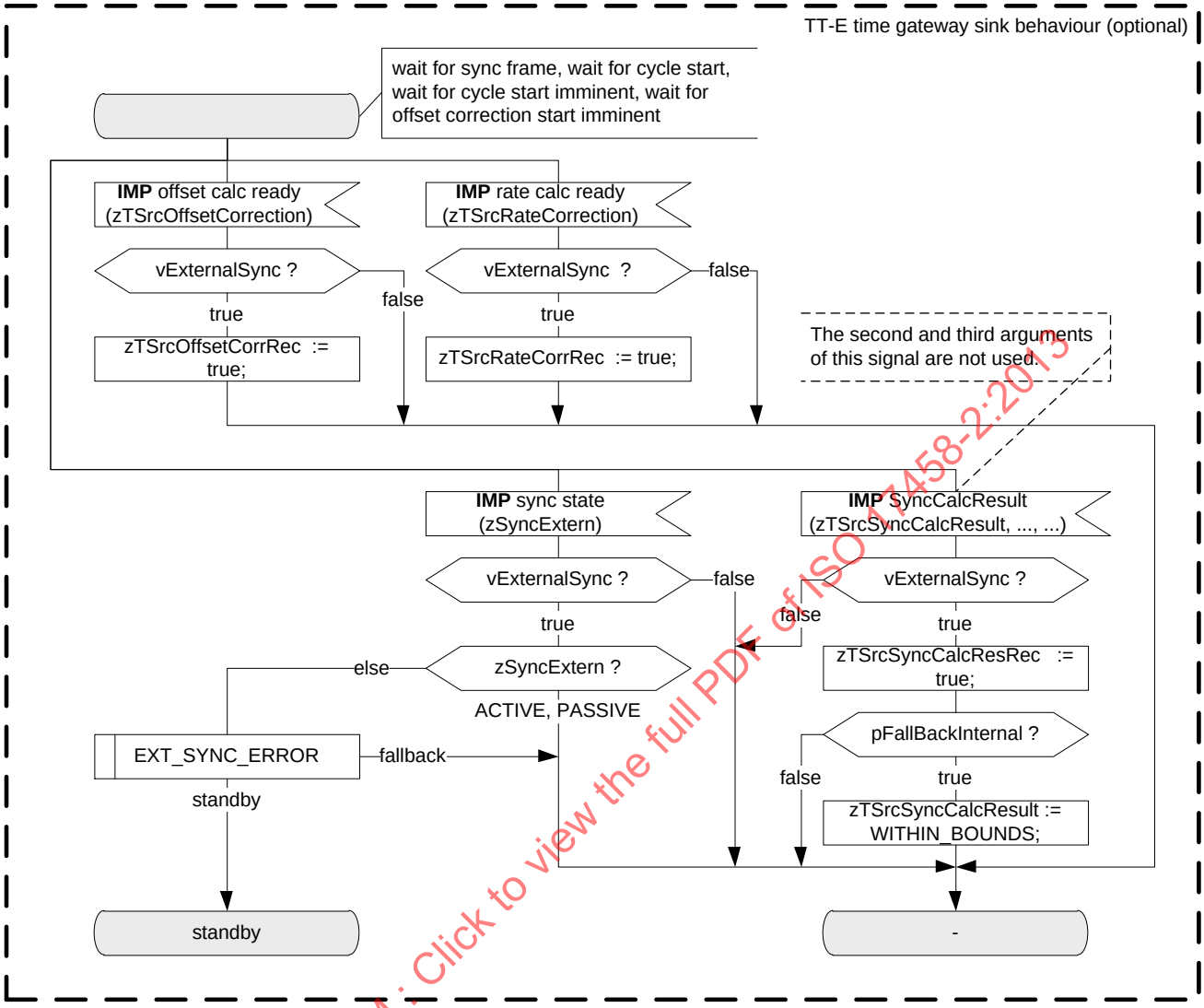



**Figure 172 — Obtain external clock sync signals [CSP]**



Should the time gateway source node indicate that it is neither in the state *POC:normal active* nor in the state *POC:normal passive*, the time gateway sink node will behave as if it has lost synchronisation to the time gateway source node (see Figure 175).

If the time gateway sink node is the sole provider of startup frames of the cluster, it shall not switch to *POC:normal passive*.

To that end, the node overwrites the *zSyncCalcResult* value of the time gateway source node with the fault free value "WITHIN_BOUNDS" if it is configured to "fall back" to the local synchronisation method, which implies that it is the sole coldstart node. If several TT-E coldstart nodes are present in the cluster, it is obviously important that they all agree upon the schedule.

Should the time gateway source node of one of the time gateway sink nodes have a problem with its clock synchronisation, it cannot ascertain the validity of its view on the time source cluster schedule.

Therefore, its time gateway sink node should (also) enter *POC:normal passive* and only return to *POC:normal active* when its time gateway source node has recovered its synchronisation with the time source cluster.

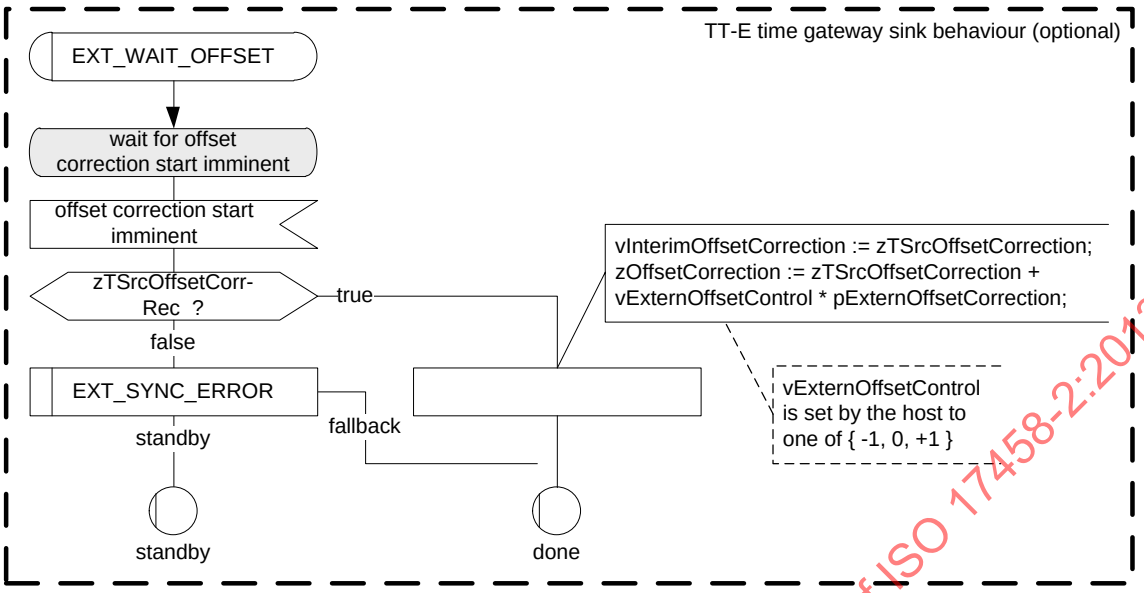



**Figure 173 — Macro EXT_WAIT_OFFSET [CSP]**

The time gateway sink node requires an update of the offset correction term before it enters the offset correction phase. Figure 173 shows how it checks just before the start of the offset correction phase whether it has received a new offset correction term from its time gateway source node. If it has not received an update, it will behave as if it has lost synchronisation to the time gateway source node (see Figure 175).

If it has received an update, it will discard its locally generated offset correction term and use the one supplied by the time gateway source node instead.

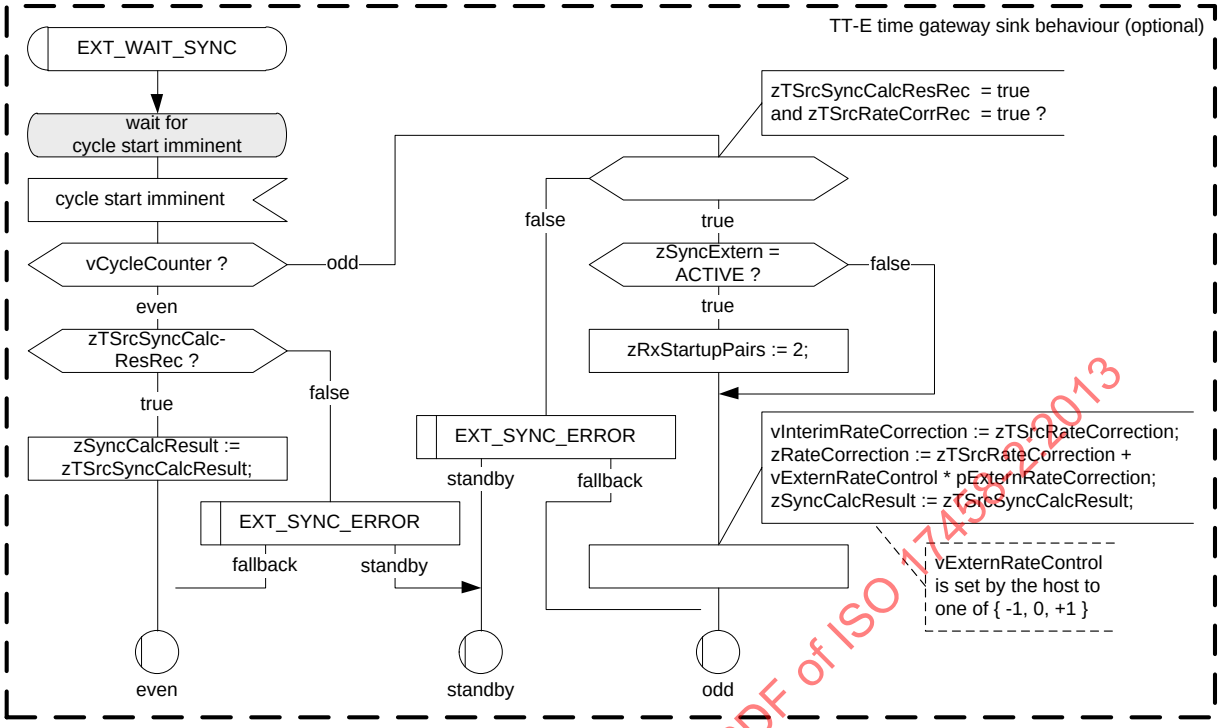



**Figure 174 — Macro EXT_WAIT_SYNC [CSP]**

The time gateway sink node requires an update of the rate correction term before it enters a new even cycle.

Figure 174 shows how it checks just before the start of the new even cycle whether it was provided with a new rate correction term by its time gateway source node. If it has not received such an update, it will behave as if it has lost synchronisation to the time gateway source node (see Figure 175).

If the time gateway sink node has received an update, it will discard its locally generated rate correction term and use the one supplied by the time gateway source node instead. In addition, if the *zSyncExtern* variable indicates that the time gateway source node is in *POC:normal active* the time gateway sink will set the variable *zRxStartupPairs* to 2 - this allows the time gateway sink to follow the time gateway source as it makes a transition from *POC:normal passive* to *POC:normal active*.

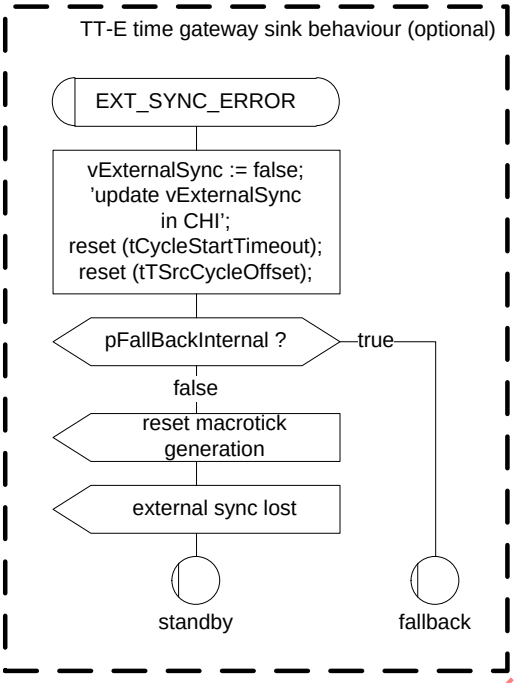



**Figure 175 — Macro EXT_SYNC_ERROR [CSP]**

The macro EXT_SYNC_ERROR describes the behaviour of the time gateway sink node in the case that it determines a loss of synchronisation with the time gateway source node. In such a case, the time gateway sink node first informs the CHI. The CSP then checks whether it is configured to "fall back" to the local synchronisation method. If yes, it does so; if not, it stops the macrotick generation process and indicates the loss of synchronisation to the POC, which will then enter *POC:halt*.

## 12.7 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 (see Figure 176) generates (corrected) macroticks. There are three different ways to begin the process of generating macroticks.

— For a leading coldstart node, the protocol operation control (POC) process initiates macrotick generation (via the *coldstart* signal) if the conditions to start the node as a coldstart node are satisfied, or

— For an integrating or following coldstart node, the clock synchronisation startup (CSS) processes initiate macrotick generation (via the *start macrotick generation* signal) upon the reception of an acceptable pair of startup frames, or

— For a TT-E coldstart node, the clock synchronisation process (CSP) initiates macrotick generation (via the *start macrotick generation* signal) if the conditions to perform the startup of the time sink cluster are satisfied.

Either of the three 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.

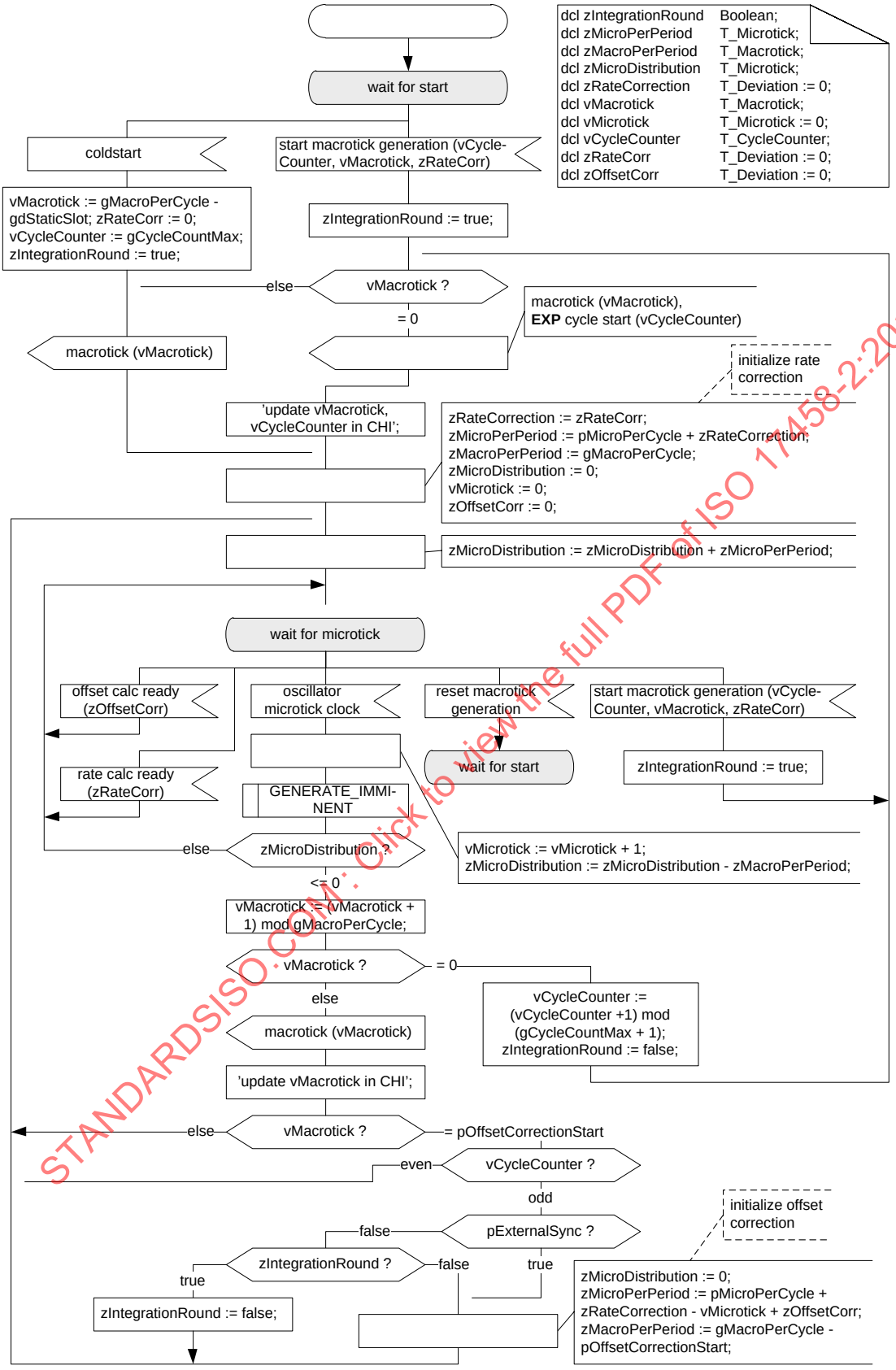





**Figure 176 — Macrotick generation [MTG]**

Figure 177 depicts the macro GENERATE_IMMINENT [MTG].

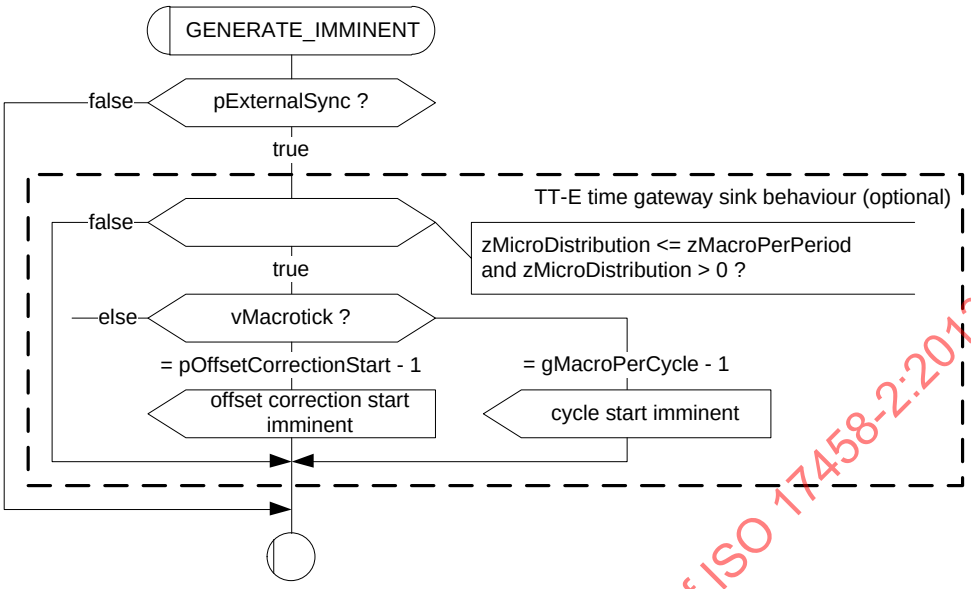



**Figure 177 — Macro GENERATE_IMMINENT [MTG]**

The macro GENERATE_IMMINENT encapsulates the generation of two signals used only by TT-E coldstart nodes. These signals are used to determine the points in time at which a TT-E coldstart node decides that it has lost synchronisation with its time gateway source if it has not received the appropriate clock correction value. The offset correction value should be received before the *offset correction start imminent* signal and the rate correction value should be received before the *cycle start imminent* signal, otherwise the TT-E coldstart node will assume a loss of synchronisation.

Figure 178 depicts the termination of the MTG process [MTG].

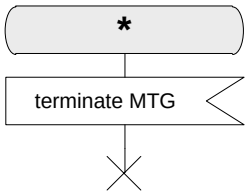



**Figure 178 — Termination of the MTG process [MTG]**



## 12.8 Sync frame configuration

### 12.8.1 Configuration rules

FlexRay supports a distributed clock synchronisation that can be configured in many different ways. Table 7 specifies a number of rules constraining the possible configurations.

**Table 7 — Configuration rules for the clock synchronisation**

| Element | limit, range | | |
|---|---|---|---|
| | TT-D | TT-L | TT-E |
| Number of sync nodes | 2[a]... *cSyncFrameIDCountMax* | 1 | 1 ... **floor**( *cSyncFrameIDCountMax* / 2 ) |
| Number of sync frame ID's | number of sync nodes | 2 | = 2 * number of sync nodes |
| Number of static slots | >= number of sync frame ID's | >=2 | >= number of sync frame ID's |

[a] A TT-D cluster with only two sync nodes (each of which is also configured to be a coldstart node) would be able to start up and continue to operate, but a loss of either of the two sync nodes would result in the loss of all communication.

### 12.8.2 TT-D cluster

A TT-D cluster consists of a number of sync nodes and an arbitrary number of non-sync nodes. Each coldstart node is always a sync node but there can be more sync nodes than coldstart nodes. A number of nodes shall be configured as sync nodes depending on the following rules.

— At least two nodes shall be configured to be sync nodes.

— At least two of the sync nodes shall also be TT-D coldstart nodes.

— At most *cSyncFrameIDCountMax* nodes shall be configured to be sync nodes.

— Only nodes with *pChannels* = *gChannels* may be sync nodes (i.e., sync nodes shall be connected to all configured channels).

— Sync nodes that support two channels shall send two sync frames in the static segment, one on each channel in the corresponding slot with the slot number which equals *pKeySlotID*[121]. Sync nodes that only support a single channel shall send one sync frame in the static segment. The sync frame shall be sent in the slot with the slot number which equals *pKeySlotID*.

— The sync frames shall be sent in all slots with the same slot number, i.e. in each cycle.

— Non-sync nodes shall not transmit frames with the sync frame indicator set to one.

### 12.8.3 TT-E cluster

A TT-E cluster consists only of coldstart nodes and an arbitrary number of non-sync nodes. In that respect the following rules shall be observed.

— At least one node shall be configured to be a sync node.

— Each sync node shall also be a TT-E coldstart node.

— At most *cSyncFrameIDCountMax* / 2 nodes shall be configured to be sync nodes.

— Only nodes with *pChannels* = *gChannels* may be sync nodes (i.e., sync nodes shall be connected to all configured channels).

121) The frames sent on the two channels in these slots do not need to be identical (i.e., they may have differing payloads), but they shall both be sync frames.

— Sync nodes that support two channels shall send four sync frames in the static segment, two on each channel, in the slots with slot number equal to *pKeySlotID* or *pSecondKeySlotID*[122]. Sync nodes that support only a single channel shall send two sync frames in the static segment in the slots with slot number equal to *pKeySlotID* or *pSecondKeySlotID*.

— The sync frames shall be sent in all slots with the same slot number, i.e. in each cycle.

— Non-sync nodes shall not transmit frames with the sync frame indicator set to one.

### 12.8.4 TT-L cluster

A TT-L cluster consists only of a single coldstart node and an arbitrary number of non-sync nodes. In that respect the following rules shall be observed.

— A single node shall be configured to be a sync node.

— The single sync node shall be a TT-L coldstart node.

— Only nodes with *pChannels* = *gChannels* may be sync nodes (i.e., sync nodes shall be connected to all configured channels).

— Sync nodes that support two channels shall send four sync frames in the static segment, two on each channel, in the slots with slot number equal to *pKeySlotID* or *pSecondKeySlotID*[123]. Sync nodes that support only a single channel shall send two sync frames in the static segment in the slots with slot number equal to *pKeySlotID* or *pSecondKeySlotID*.

— The sync frames shall be sent in all slots with the same slot number, i.e. in each cycle.

— Non-sync nodes shall not transmit frames with the sync frame indicator set to one.

## 12.9 Time gateway interface

The time gateway interface connects two communication controllers. It provides information about the schedule of the time gateway source node to the time gateway sink node.

The signals to be provided by the time gateway source node are marked in the SDL description using the EXP keyword. The signals in question are as follows.

— *cycle start*
This signal is used by the time gateway sink node to initialize its own schedule and later on to monitor if it still operates synchronously to the time gateway source node.

— *sync state*
This signal is used by the time gateway sink node to determine if the time gateway source node is in *POC:normal active*, *POC:normal passive* or a different state.

— *offset calc ready*
This signal is used by the time gateway sink node to perform the same clock offset correction as the time gateway source node.



122) The frames sent in these slots by a TT-E sync node do not need to be identical (i.e., different payloads may be sent on each channel, or in each of the key slots), but all shall be sync frames.
123) The frames sent in these slots by a TT-L sync node do not need to be identical (i.e., different payloads may be sent on each channel, or in each of the key slots), but all shall be sync frames.

— *rate calc ready*
This signal is used by the time gateway sink node to perform the same clock rate correction as the time gateway source node.

— *SyncCalcResult*
This signal is used by the time gateway sink node to determine if the time gateway source node maintains synchronisation with the time source cluster.

The fixed offset of *cdTSrcCycleOffset* microticks between the schedule of the time gateway source node and the time gateway sink node ensures that each transferred piece of information arrives in time to be properly used by the time gateway sink node.

# 13 Controller Host Interface

## 13.1 Principles

The controller host interface (CHI) manages the data and control flow between the host processor and the FlexRay protocol engine within each node[124)]. 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 179.

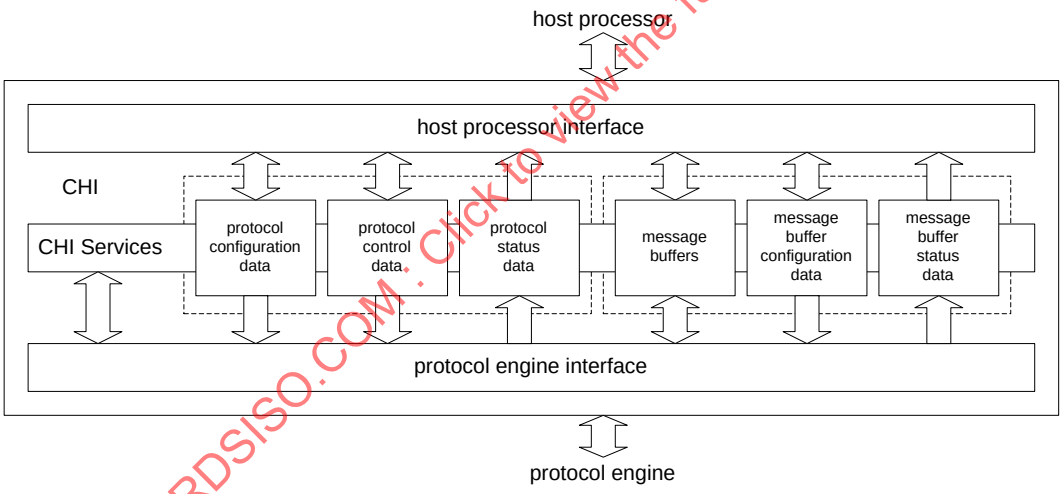



**Figure 179 — Conceptual architecture of the controller host interface**

The protocol data interface manages the protocol configuration data, the protocol control data, and the protocol status data. The message data interface manages the message buffers, the message buffer configuration data, and the message buffer status data.

In addition, the CHI provides a set of CHI services that define self-contained functionality that is transparent to the operation of the protocol.

124) Due to implementation constraints the CHI may add product specific delays for data or control signals exchanged between the host and the protocol engine.

The descriptions of the CHI in this clause are behavioural 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 behaviour as the description, but it need not have the same underlying structure or mechanisms.

## 13.2 Description

The relationships between the CHI and the other protocol processes are depicted in Figure 180[125)].

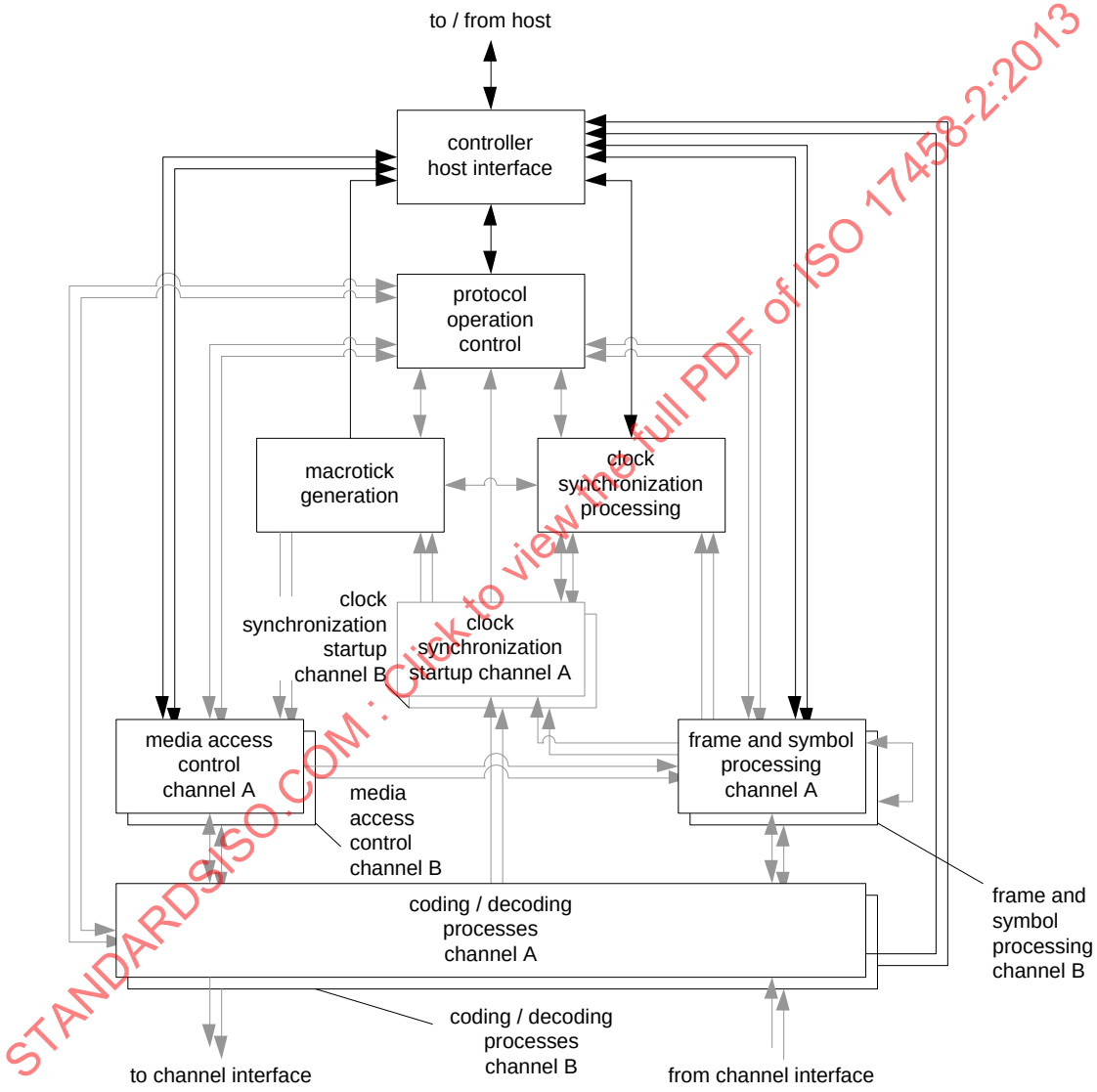





**Figure 180 — Controller host interface context**

125) The dark lines represent data flows between mechanisms that are relevant to this subclause. The lighter gray lines are relevant to the protocol, but not to this clause.

## 13.3 Interfaces

### 13.3.1 Protocol data interface

#### 13.3.1.1 Protocol configuration data

##### 13.3.1.1.1 Host read and write access

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.

##### 13.3.1.1.2 Communication cycle timing configuration

All configuration data relating to the following communication cycle timing parameters shall be treated as protocol configuration data:

— the number of microticks *pMicroPerCycle* constituting the duration of the communication cycle;

— the number of macroticks *gMacroPerCycle* within a communication cycle;

— the number of static slots *gNumberOfStaticSlots* in the static segment;

— the number of macroticks *gdStaticSlot* constituting the duration of a static slot within the static segment;

— the number of macroticks *gdActionPointOffset* constituting the offset of the action point within static slots;

— the number of macroticks *gdMinislot* constituting the duration of a minislot;

— the number of minislots *gNumberOfMinislots* within the dynamic segment;

— the number of macroticks *gdMinislotActionPointOffset* constituting the offset of the action point within a minislot of the dynamic slot;

— the number of minislots *gdDynamicSlotIdlePhase* constituting the duration of the idle phase within a dynamic slot;

— the number of the last minislot *pLatestTx* in which transmission can start in the dynamic segment;

— the number of macroticks *gdSymbolWindow* constituting the duration of the symbol window;

— the number of macroticks *gdSymbolWindowActionPointOffset* constituting the offset of the action point within the symbol window;

— the cycle number *gCycleCountMax* after which the cycle counter is reset back to zero[126)].

In addition to the above, an implementation shall allow a system designer control over the parameters *gdSampleClockPeriod* and *pSamplesPerMicrotick*. These parameters need not be directly implemented, but may instead be derived from other parameters not explicitly defined in this specification.

For example, *gdSampleClockPeriod* may be derived from parameters that set the prescalers, phase-locked loop multipliers, etc. for an implementation's sample clock and the designer's knowledge of the design frequency of the underlying clocks. As a result, there is no specific requirement to be able to read or configure

126) The cycle counter will return to the same value after *gCycleCountMax* + 1 cycles.

these exact parameters, but there is a requirement for a system designer to be able to control their values, i.e., to control the period of the sample clock and the duration of the microtick to allow the values for those quantities required by the specification.

#### 13.3.1.1.3 Protocol operation configuration

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

— the number of consecutive even / odd cycle pairs with missing clock correction terms *gMaxWithoutClockCorrectionFatal* that will cause the protocol to transition from the *POC:normal active* or *POC:normal passive* state into the *POC:halt* state;

— the number of consecutive even / odd cycle pairs with missing clock correction terms *gMaxWithout ClockCorrectionPassive* that will cause the protocol to transition from the *POC:normal active* to the *POC:normal passive* state;

— the number of microticks *pClusterDriftDamping* constituting the cluster drift damping factor used for rate correction within clock synchronisation;

— the number of macroticks *pOffsetCorrectionStart* between the start of the communication cycle and the start of the offset correction within the NIT;

— the number of microticks *pExternOffsetCorrection* constituting the correction term used to correct the calculated offset correction value in the course of external clock synchronisation;

— the number of microticks *pExternRateCorrection* constituting the correction term used to correct the calculated rate correction value in the course of external clock synchronisation;

— the number of microticks *pOffsetCorrectionOut* constituting the upper bound for a permissible offset correction;

— the number of microticks *pRateCorrectionOut* constituting the upper bound for a permissible rate correction and the maximum drift offset between two nodes operating with non-synchronized clocks for one communication cycle,

— the Boolean parameter *pAllowHaltDueToClock* that controls the transition to the *POC:halt* state due to a clock synchronisation error;

— the number of consecutive even / odd cycle pairs *pAllowPassiveToActive* during which valid clock synchronisation terms shall 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;

— the Boolean parameter *pKeySlotOnlyEnabled* that defines whether, after completing startup, a node is restricted to send only in its key slots or is allowed to transmit in all assigned slots;

— the Boolean parameter *pKeySlotUsedForStartup* that defines whether the specific node shall send startup frames;

— the Boolean parameter *pKeySlotUsedForSync* that defines whether the specific node shall send sync frames;

— the slot ID of the key slot, *pKeySlotID*. A node that does not have a key slot would configure *pKeySlotID* to zero (a value which will never match any actual static slot ID). The effect of such a zero configuration for *pKeySlotID* is that no static slot has the characteristics of the key slot (and thus the node, in effect, does not have a key slot). The key slot, if the node has one, shall be assigned to the node by the CHI in all cycles. In particular, the slot ID indicated by a non-zero configuration of *pKeySlotID* is assigned to the

node (in the sense that for that slot the variable *vTCHI!Assignment* is set to ASSIGNED) on all configured channels in all communication cycles;

— the slot ID of the second key slot, *pSecondKeySlotID* in which a second startup frame shall be sent when operating as coldstart node in a TT-L or TT-E cluster;

— the number of microticks *pDecodingCorrection* used by the node to calculate the primary time reference point;

— the enumeration *pChannels* that indicates the channels to which the node is connected. The configuration of channels supported by the device, *pChannels*, has a unique characteristic in that it may affect significant hardware details of the implementation. Dual channel devices shall be able to support single channel operation and single channel devices shall be configurable to work on either channel A or channel B. As a result, it is required that an implementation be able to configure *pChannels*, but the ability to configure this more than once while the CC is in the power on state (see 6.1.2) is not required. This mechanism does not need to be available *in POC:config*. A device may not, however, allow this parameter to be modified in the *POC:ready*, *POC:normal active*, *POC:normal passive*, or *POC:halt* states, or in any of the states that are defined in the WAKEUP and STARTUP macros of the POC process;

— the maximum number of consecutive low bits *gdCASRxLowMax* which the node would accept as a valid CAS symbol;

— the number of bits *gdIgnoreAfterTx* for which bit strobing is paused after a transmission;

— the number of microticks *pDelayCompensation[A]* used to include the channel A specific reception delay in the calculation of the primary time reference point;

— the number of microticks *pDelayCompensation[B]* used to include the channel B specific reception delay in the calculation of the primary time reference point;

— the number of two-byte words *gPayloadLengthStatic* contained in the payload segment of a static frame;

— the number of bits *gdTSSTransmitter* within the transmission start sequence;

— the maximum number *gSyncFrameIDCountMax* of distinct sync frame identifiers that may be present in a given cluster;

— the optional[127] Boolean parameter *pFallBackInternal* that defines whether a time gateway sink node will switch to local clock operation when synchronisation with the time gateway source node is lost (*pFallBackInternal* = true) or will instead go to *POC:halt* (*pFallBackInternal* = false)[128];

— the optional[129] Boolean parameter *pExternalSync* that defines whether the node begins operation[130] with the clock synchronisation externally synchronized (*pExternalSync* = true) or not externally synchronized (*pExternalSync* = false);

— the Boolean parameter *pTwoKeySlotMode* that defines whether the node operates as a coldstart node in a TT-E or TT-L cluster.

#### 13.3.1.1.4 Wakeup and startup configuration

All configuration data relating to the following wakeup and startup parameters shall be treated as protocol configuration data:

127) The Boolean parameter is only required in implementations that implement a time gateway sink.
128) This parameter shall not be set to true if the cluster contains more than one TT-E coldstart node.
129) The Boolean parameter is only required in implementations that implement a time gateway sink.
130) It is possible that the actual synchronisation mode of the cluster (as represented by the variable *vExternalSync*) can change from externally synchronized to not externally synchronized during the operation of the cluster.

— the number of bits *gdWakeupRxIdle* used by the node to test the duration of the received idle or 'active high' parts of a wakeup;

— the number of bits *gdWakeupRxLow* used by the node to test the duration of the received 'active low' parts of a wakeup;

— the number of bits *gdWakeupRxWindow* used by the node to test the duration of a received wakeup;

— the number of bits *gdWakeupTxIdle* used by the node to transmit the idle part of the wakeup symbol;

— the number of bits *gdWakeupTxActive* used by the node to transmit the 'active low' part of the wakeup symbol and the 'active low' and 'active high' parts of a WUDOP;

— the number of wakeup symbols *pWakeupPattern* to be sent by the node to create a wakeup pattern;

— the enumeration *pWakeupChannel* that indicates on which channel a wakeup symbol shall be sent upon issuing the WAKEUP command;

— 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;

— 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;

— the number of microticks *pMicroInitialOffset[A]* between the secondary time reference point on channel A and the macrotick boundary immediately following the secondary time reference point;

— the number of microticks *pMicroInitialOffset[B]* between the secondary time reference point on channel B and the macrotick boundary immediately following the secondary time reference point;

— the number of microticks *pdAcceptedStartupRange* constituting the expanded range of measured deviation for startup frames during integration;

— the number of microticks *pdListenTimeout* constituting the upper limit for the startup and wakeup listen timeout;

— the maximum number of times *gColdstartAttempts* that a node is permitted to attempt to start the cluster by initiating schedule synchronisation;

— the upper limit *gListenNoise* of the number of startup and wakeup listen timeouts in the presence of noise.

#### 13.3.1.1.5 Network Management Vector configuration

All configuration data relating to the following Network Management Vector parameters shall be treated as protocol configuration data:

— the number of bytes *gNetworkManagementVectorLength* contained in the network management vector;

— the Boolean parameter *pNMVectorEarlyUpdate* that defines the segment after which the accrued network management vector is exported to the CHI;

#### 13.3.1.2 Protocol control data

##### 13.3.1.2.1 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 clause 13 and Table 4[131)]

— An ALLOW_COLDSTART command that activates the capability of the node to coldstart the cluster.

— An ALL_SLOTS command that controls the transition from the key slot only mode to the all slots transmission mode.

— A CONFIG command that causes a transition of the POC process from either the *POC:default config* state or the *POC:ready* state to the *POC:config* state.

— A CONFIG_COMPLETE command that causes a transition of the POC process from the *POC:config* state to the *POC:ready* state.

— A FREEZE command that causes a transition of the POC process from any POC state to the *POC:halt* state.

— An IMMEDIATE_READY command that causes an immediate transition of the POC process to the *POC:ready* state.

— A RUN command that initiates the startup procedure.

— A DEFAULT_CONFIG command that causes a transition of the POC process from the *POC:halt* state to the *POC:default config* state.

— A DEFERRED_HALT command that causes a transition of the POC to the *POC:halt* state.

— A WAKEUP command that initiates the wakeup procedure.

— A DEFERRED_READY command that causes a transition of the POC process to the *POC:ready* state from all states except *POC:default config*, *POC:config*, *POC:ready*, *POC:halt*.

— A CLEAR_DEFERRED command that removes any pending DEFERRED_READY or DEFERRED_HALT command.

##### 13.3.1.2.2 Control of MTS and WUDOP transmission

The CHI shall provide means for the host

— to control the transmission of MTS symbols on channel A within the symbol window. To perform this the CHI interacts with the protocol engine via the variable *vTransmitMTS_A* (as defined within the MAC_A process),

— to control the transmission of MTS symbols on channel B within the symbol window. To perform this the CHI interacts with the protocol engine via the variable *vTransmitMTS_B* (as defined within the MAC_B process),

131) 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.

— to control the transmission of WUDOPs on channel A within the symbol window. To perform this the CHI interacts with the protocol engine via the variable *vTransmitWUDOP_A*[132] (as defined within the MAC_A process), and

— to control the transmission of WUDOPs on channel B within the symbol window. To perform this the CHI interacts with the protocol engine via the variable *vTransmitWUDOP_B* (as defined within the MAC_B process).

The control of MTS transmission shall support, at a minimum, the ability of the host to request a single MTS transmission (i.e., a "one-shot" or manual MTS request).

The control of the WUDOP transmission shall support the ability of the host to request a single WUDOP transmission (i.e., a "one-shot" or manual WUDOP request), but shall also support an automatic request by which the host can identify a set of FlexRay communication cycles in which the CC will transmit a WUDOP in the symbol window without additional host interaction. This automatic mechanism shall operate in addition to the manual mechanism (i.e., a WUDOP should be sent if either the manual or automatic mechanisms, or both, indicate that a WUDOP should be sent).

The configuration of the specific cycles in which the automatic transmission of WUDOPs takes place shall support, at a minimum, the ability to define sets of communication cycles which equal the sets which can be defined using a Cycle_Repetition and a Cycle_Offset to determine the configuration:

Automatically transmit a WUDOP in the symbol window if

— *vCycleCounter* mod Cycle_Repetition = Cycle_Offset

with

— Cycle_Repetition selected from the set of {1, 2, 4, 5, 8, 10, 16, 20, 32, 40, 50, 64}[133],

— Cycle_Offset selected from the set {0 … 63} with Cycle_Offset < Cycle_Repetition.

It is not required that an implementation supports independent sets of repetition and offset parameters for each channel (i.e., a single set of parameters controlling automatic WUDOP generation for both channels is acceptable). It is acceptable, but not required, that the repetition and offset parameters controlling automatic WUDOP generation be modifiable during operation (i.e., when the node is in the *POC:normal active* state). It is required, however, that automatic WUDOP generation be independently controllable (i.e., switched on or off) on a per channel basis when the node is in the *POC:normal active* state[134].

A transition into the *POC:ready* state shall cause the CHI to reset any pending one-shot transmissions of WUDOP's or MTS's[135].

Transmissions in the symbol window shall be coordinated at a system level, i.e., the system designer shall ensure for each channel that in any given cycle at most one node will transmit either an MTS or a WUDOP in the symbol window on that channel.



132) This variable represents the control value of both the automatic and manual mechanisms for WUDOP transmission.
133) These particular values of Cycle_Repetition are those that could be achieved by combining a "power of 2" filter with a "count to 5" filter.
134) It is also allowable that automatic WUDOP generation can be switched on or off in other POC states, but the actual generation of WUDOPs will only take place when the node is in the *POC:normal active* state.
135) This prevents unintentional WUDOP or MTS transmissions left over from previous operation for TT-E coldstart nodes (for example, if an IMMEDIATE_READY command is followed by a RUN command).

#### 13.3.1.2.3 Control of external clock synchronisation

The CHI shall provide means for the host

— to control the application of the external offset correction parameter *pExternOffsetCorrection* using the control value *vExternOffsetControl*,

— to control the application of the external rate correction parameter *pExternRateCorrection* using the control value *vExternRateControl*.

A transition into the *POC:ready* state shall cause the CHI to set the variables that control the operation of the external rate and offset correction (i.e., *vExternRateControl* and *vExternOffsetControl*) such that no external rate or offset correction is requested.

### 13.3.1.3 Protocol status data

#### 13.3.1.3.1 Overview and general behaviour

The CHI shall provide the protocol status information described in the following subclauses.

The following subclauses define a number of indicators. These indicators are a type of status information that can take on two distinct values, labelled as "set" and "reset". An indicator is set (i.e., given the "set" value) by the CHI when certain events occur in the execution of the protocol, and are, in general, reset (i.e., given the "reset" value) by the host.

An implementation shall ensure that each indicator described in the following subclauses is reset under at least one of the following conditions:

— every transition into the *POC:default config* state;

— every transition into the *POC:config* state;

— every transition out of the *POC:config* state.

The specific behaviour of an implementation with respect to the reset behaviour of each indication is implementation dependent.

#### 13.3.1.3.2 Protocol operation control status

The following protocol operation control status variables shall be provided in the CHI:

— the status variable *vPOC!State* (as maintained by the POC process);

— the flag *vPOC!Freeze* (as maintained by the POC process);

— the flag *vPOC!CHIReadyRequest* (as maintained by the POC process);

— the flag *vPOC!CHIHaltRequest* (as maintained by the POC process);

— the flag *vPOC!ColdstartNoise* (as maintained by the POC process);

— the status variable *vPOC!SlotMode* (as maintained by the POC process);

— the status variable *vPOC!ErrorMode* (as maintained by the POC process);

— the number of consecutive even / odd cycle pairs *vAllowPassiveToActive* that have passed with valid rate and offset correction terms, but with the node still in *POC:normal passive* state due to a host configured delay to *POC:normal active* state (as maintained by the POC process);

— the status variable *vPOC!WakeupStatus* (as maintained by the POC process);

— the status variable *vPOC!StartupState* (as maintained by the POC process);

— the optional[136)] flag *vExternalSync* (as maintained by the CSP process).

#### 13.3.1.3.3 Wakeup and startup status

The number of remaining coldstart attempts *vRemainingColdstartAttempts* (as maintained by the POC process) shall be provided in the CHI as a startup status variable.

The CHI shall provide indicators for the following wakeup and startup events:

— 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;

— a wakeup pattern received indicator for channel A that shall be set upon 'set wakeup received indicator on A in CHI' (in accordance with the FSP_A process) and reset under control of the host;

— a wakeup pattern received indicator for channel B that shall be set upon 'set wakeup received indicator on B in CHI' (in accordance with the FSP_B process) and reset under control of the host.

#### 13.3.1.3.4 Communication cycle timing status

The following communication cycle timing variables shall be provided in the CHI:

— the macrotick *vMacrotick* (as maintained by the MTG process);

— the cycle counter *vCycleCounter* (as maintained by the MTG process);

— the slot counter *vSlotCounter* for channel A (as maintained by the MAC_A process);

— the slot counter *vSlotCounter* for channel B (as maintained by the MAC_B process).

The values provided to the host by the CHI for *vMacrotick*, *vCycleCounter* and *vSlotCounter* for channel A and B shall be valid during the states *POC:normal active* and *POC:normal passive*.

A snapshot of the following communication cycle timing variables shall be provided in the CHI:

— the rate correction value *vInterimRateCorrection* (in accordance with the CSP process):

  — if *vExternalSync* [137)] is false the rate correction value is based on the internally calculated values, is not limited by *pRateCorrectionOut*, and does not include any external rate correction value;

  — if *vExternalSync* is true the rate correction value is imported from the time gateway source and includes the limitation and external rate correction values as configured in the time gateway source.

— the offset correction value *vInterimOffsetCorrection* (as maintained by the CSP process):

136) The status variable is only required in implementations that implement a time gateway sink.

137) If the optional flag *vExternalSync* does not exist the communication controller shall behave as if *vExternalSync* is equal to false. Refer to 5.6.4.

— if *vExternalSync* is false the offset correction value is based on the internally calculated values, is not limited with *pOffsetCorrectionOut*, and does not include any external offset correction value;

— if *vExternalSync* is true the offset correction value is imported from the time gateway source and includes the limitation and external offset correction values as configured in the time gateway source.

The CHI shall provide indicators for the following communication cycle timing events:

— a sync frame overflow indicator that shall be set upon 'set sync frame overflow indicator in CHI' (in accordance with the CSP process) and reset under control of the host;

— a *pLatestTx* violation status indicator for channel A that shall be set upon 'set *pLatestTx* violation status indicator on A in CHI' (in accordance with the MAC_A process), and reset under control of the host;

— a *pLatestTx* violation status indicator for channel B that shall be set upon 'set *pLatestTx* violation status indicator on B in CHI' (in accordance with the MAC_B process), and reset under control of the host;

— a transmission across boundary violation status indicator for channel A that shall be set upon 'set transmission across slot boundary violation indicator on A in CHI' (in accordance with the FSP_A process);

— a transmission across boundary violation status indicator for channel B that shall be set upon 'set transmission across slot boundary violation indicator on B in CHI' (in accordance with the FSP_B process).

#### 13.3.1.3.5 Synchronisation frame status

A snapshot of the following information, derived from the *vsSyncIDListA* and *vsSyncIDListB* variables provided by the CSP process, shall be provided in the CHI:

— 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;

— 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;

— 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;

— 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 This implies that for some NIT configurations the data update may complete after the start of the next cycle.

The number of consecutive even / odd cycle pairs *vClockCorrectionFailed* that have passed without clock synchronisation having performed an offset or a rate correction due to lack of synchronisation frames (as maintained by the POC process) shall be provided in the CHI.

#### 13.3.1.3.6 Startup frame status

The number of channel aligned startup frame pairs received or transmitted during the previous double cycle, aggregated across both channels, derived from the *vStartupPairs* variable provided by the CSP process, shall be provided in the CHI as a snapshot.

The availability of this information depends on the synchronisation method and role of the node in the cluster. For TT-E coldstart nodes the information shall be updated no sooner than one microtick before the end of the NIT in the odd cycle and no later than 10 macroticks after the start of the next cycle. For all other nodes the information shall be updated no sooner than the start of the NIT in the odd cycle and no later than 10 macroticks after the start of the offset correction phase of the NIT.

NOTE This implies that for some NIT configurations the data update may complete after the start of the next cycle.

#### 13.3.1.3.7 Symbol window status

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

— the flag *vSS!ValidMTS* for channel A (in accordance with the FSP_A process);

— the flag *vSS!ValidMTS* for channel B (in accordance with the FSP_B process);

— the flag *vSS!SyntaxError* for channel A (in accordance with the FSP_A process);

— the flag *vSS!SyntaxError* for channel B (in accordance with the FSP_B process);

— the flag *vSS!BViolation* for channel A (in accordance with the FSP_A process);

— the flag *vSS!BViolation* for channel B (in accordance with the FSP_B process);

— the flag *vSS!TxConflict* for channel A (in accordance with the FSP_A process);

— the flag *vSS!TxConflict* for channel B (in accordance with the FSP_B process);

The following list indicates the behaviour of the CHI depending on the activities detected on the RxD input.

— The low phases of a WUDOP might cause an indication of a valid MTS even if only a WUDOP was actually present. This may or may not happen (depending on the data rate, which affects the duration of what is accepted as a valid MTS).

— Because the low phases of a WUDOP can be considered as a valid MTS, the reception of multiple valid MTS's within a symbol window is not indicated as a syntax error.

— Although the WUDOP is transmitted in the symbol window, an indication that a wakeup has been received may not occur until sometime during the NIT (i.e., the symbol window status may become available to the host before the indication that a wakeup has been received becomes available to the host).

— In the normal situation, the reception of an MTS or WUDOP will not result in boundary violations or syntax errors (i.e., those conditions represent exceptional circumstances such as additional communication elements, noise on the link, etc.).

— It is possible to detect a valid MTS, or to detect a wakeup, even in the presence of syntax errors or boundary violations

#### 13.3.1.3.8 NIT status

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

— the flag *vSS!SyntaxError* for channel A (in accordance with the FSP_A process);

— the flag *vSS!SyntaxError* for channel B (in accordance with the FSP_B process);

— the flag *vSS!BViolation* for channel A (in accordance with the FSP_A process);

— the flag *vSS!BViolation* for channel B (in accordance with the FSP_B process).

#### 13.3.1.3.9 Aggregated channel status

The aggregated channel status provides the host with an accrued status of channel activity for all communication slots regardless of whether they are assigned for transmission or subscribed for reception. The status is aggregated over a period that is freely definable by the host.

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

— a channel specific valid frame indicator for channel A that shall be set if a valid frame was received in any static or dynamic slot on channel A (i.e., one or more static or dynamic slots had *vSS!ValidFrame* equal to true) and reset under control of the host;

— a channel specific valid frame indicator for channel B that shall be set if a valid frame was received in any static or dynamic slot on channel B (i.e., one or more static or dynamic slots had *vSS!ValidFrame* equal to true) and reset under control of the host;

— 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;

— 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;

— 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;

— 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;

— 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;

— 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;

— a channel specific slot boundary violation indicator for channel A that shall be set if one or more slot boundary violations were observed on channel A (i.e., one or more static or dynamic slots including symbol window and NIT had *vSS!BViolation* equal to true) and reset under control of the host;

— a channel specific slot boundary violation indicator for channel B that shall be set if one or more slot boundary violations were observed on channel B (i.e., one or more static or dynamic slots including symbol window and NIT had *vSS!BViolation* equal to true) and reset under control of the host;

— a channel specific transmission conflict indicator for channel A that shall be set if one or more transmission conflicts were observed on channel A (i.e., one or more static or dynamic slots and / or the symbol window had *vSS!TxConflict* equal to true) and reset under control of the host;

— a channel specific transmission conflict indicator for channel B that shall be set if one or more transmission conflicts were observed on channel B (i.e., one or more static or dynamic slots and / or the symbol window had *vSS!TxConflict* equal to true) and reset under control of the host.

#### 13.3.1.3.10 Dynamic segment status

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

— a channel specific value which represents the slot counter of the last frame transmitted by the node on channel A in the dynamic segment as reflected by the variable *vLastDynTxSlot* in the MAC_A process. It is updated at the end of the dynamic segment and would have a value of zero if no frame was transmitted during the dynamic segment[138)];

— a channel specific value which represents the slot counter of the last frame transmitted by the node on channel B in the dynamic segment as reflected by the variable *vLastDynTxSlot* in the MAC_B process. It is updated at the end of the dynamic segment and would have a value of zero if no frame was transmitted during the dynamic segment;

— a channel specific dynamic resynchronisation attempted flag as reflected by the variable *vDynResync Attempt* in the MAC_A process which is set if either a slot was skipped, or a transmission might have been blocked (i.e., the *zNoTxSlot* variable was set to true, whether or not this actually resulted in one less transmission in the system), otherwise it is reset. It is updated at the end of the dynamic segment;

— a channel specific dynamic resynchronisation attempted flag as reflected by the variable *vDynResync Attempt* in the MAC_B process which is set if either a slot was skipped, or a transmission might have been blocked (i.e., the *zNoTxSlot* variable was set to true, whether or not this actually resulted in one less transmission in the system), otherwise it is reset. It is updated at the end of the dynamic segment.

### 13.3.2 Message data interface

#### 13.3.2.1 Subject

The message data interface addresses

— the management of the communication slots in which the node shall transmit messages,

— the subscription of messages the host wants to receive, and

— the exchange of message data between the host and the protocol engine within the node.

Message transmission pertains to the message data flow from the host out to a FlexRay network, and message reception pertains to the message data flow from a FlexRay network in to the host.

#### 13.3.2.2 Communication slot assignment

A node in the *POC:normal active* state is able to transmit messages by sending frames on one or both channels of the FlexRay bus and is able to receive messages by receiving frames on one or both channels of the FlexRay bus.

---

138) This variable can be used to determine if the frame corresponding to a given slot in the dynamic segment was actually transmitted. This is especially beneficial for continuous transmission mode since then the "frame transmitted" flag is less useful. In the event of an aborted transmission due to a *pLatestTx* violation in the dynamic segment, this value will indicate the ID of the frame that was aborted.

The information on when a node shall send or receive a message is called communication slot assignment.

Communication slot assignment shall be done for each available channel.

A specific communication slot of a specific communication cycle can be identified by the pair of a slot number and a communication cycle number in either the static or the dynamic segment.

Slot multiplexing, i.e. assigning slots having the same slot identifier but with different communication cycle numbers to different nodes, is allowed by the protocol in the static segment for slots which are not configured as key slots. The configuration of assignment shall ensure that transmission in the cluster is conflict-free[139]. Slot multiplexing is allowed for all slots in the dynamic segment. It is up to the application or the configuration to ensure that transmission in the cluster is conflict-free.

#### 13.3.2.3 Communication slot assignment for transmission

##### 13.3.2.3.1 General behaviour

A specific TDMA slot (communication slot) in either the static or dynamic segment of a specific communication cycle shall be assigned to a node for transmission by assigning the corresponding slot identifier and communication cycle number for a channel to the node according to the constraints listed in subclauses 9.1.4.1 and 9.1.5.1.

A node in the *POC:normal active* state will always transmit a null frame or a non-null (data) frame in a slot assigned to this node for transmission within the static segment. A node in the *POC:normal active* state will always transmit a non-null (data) frame in a slot assigned to this node for transmission within the dynamic segment if an active transmit message buffer is found (see 13.3.2.7.6).

##### 13.3.2.3.2 Cycle-independent and cycle-dependent slot assignment

The CHI shall provide the possibility to assign communication slots of a channel to the node for transmission independent of the cycle number, i.e. all slots sharing the slot ID in all communication cycles are assigned to this node on this channel. This method of assignment is referred to as "cycle-independent slot assignment".

The CHI shall provide the possibility to assign individual slots (identified by the pair of a slot number and a cycle counter number) or sets of slots (identified by a slot number and a set of communication cycle numbers) of a channel to the node for transmission. This method of assignment is referred to as "cycle-dependent slot assignment". The CHI shall provide a mechanism to enable or disable the possibility of cycle-dependent slot assignment for slots located in the static segment[140]. The enable / disable mechanism for cycle-dependent slot assignment shall be available to the host only while the node is in the *POC:config* state.

If cycle-dependent slot assignment is disabled, the CHI shall only allow cycle-independent slot assignment for communication slots in the static segment. In this case the node will transmit a frame in all slots of the static segment sharing the slot number of the assigned slot in all communication cycles.

In the static segment when a slot in a communication cycle occurs and this slot is assigned to a node, the node shall transmit either a non-null frame or a null frame in that slot. Specifically, a null frame will be sent if there is no data ready, or if there is no transmit buffer configured for this slot (see 13.3.2.8.2).

In the dynamic segment when a slot in a communication cycle occurs and this slot is assigned to a node, the node only transmits a non-null frame in this slot if an active transmit message buffer is found (see 13.3.2.7.6.

---

139) It is also possible to achieve slot multiplexing in the static segment by reconfiguring assignment during the operation in the system. In this case it is up to the application to ensure conflict-free transmission.

140) The enable / disable mechanism allows the user to select between static segment behaviour consistent with the previous versions of the protocol (where a node that transmits in a slot with a specific slot number in any cycle shall transmit either a non-null frame or a null frame in all slots with this number in all cycles) and new static segment behaviour that allows a node to transmit in a slot with a specific slot number in only some cycles (i.e., it is no longer necessary that a node that transmits in a slot in a particular cycle shall transmit in all slots with this number).

In both segments the system designer shall ensure that no two nodes transmit in the same slot.

#### 13.3.2.3.3 Transmission slot assignment list

The set of all slots assigned to a node for transmission is called the "transmission slot assignment list".

An implementation may explicitly implement the transmission slot assignment list or derive it dynamically from the transmit buffer assignment (see 13.3.2.8.2) or implement a mixture of explicit and implicit assignment.

In any case the implementation shall ensure that all requirements for the transmission slot assignment list in this subclause are fulfilled. This is especially true for the supported sets of slots which can be assigned to the node for transmission. The transmission slot assignment list has to support all combinations of sets of slots for which transmit buffers can be configured.

The CHI has to ensure that at the beginning of each slot it provides up to date information to the protocol engine as described in 13.3.2.8.3.

In the static segment this is especially true for the header CRC. If there is no transmit buffer configured for a specific entry of the transmission slot assignment list, the corresponding header CRC for this specific transmission slot assignment list entry shall be part of the transmission slot assignment list so that a valid null frame can be sent. Further descriptions of the mechanisms in this subclause assume that the transmission slot assignment list includes the corresponding header CRC for each entry.

An implementation shall provide the ability to modify the transmission slot assignment list in all states other than *POC:default config*, and shall prevent modification of the transmission slot assignment list in the *POC:default config* state.

An implementation shall provide a configurable mechanism to prevent the host from modifying the transmission slot assignment list in states other than *POC:config*. Configuration (i.e., enabling or disabling) of this mechanism shall only be possible in the *POC:config* state. It is allowed, but not required, that this configuration mechanism be the same mechanism that allows configuration of write access to the message buffer configuration information described in 13.3.2.6.2.

Modification of the transmission slot assignment list shall only be possible in states other than *POC:config* if this capability was explicitly enabled during the *POC:config* state.

#### 13.3.2.3.4 Key slot assignment

For key slots only cycle-independent slot assignment is allowed. Key slots shall be assigned to a node for all connected channels, i.e. all slots of all connected channels sharing the key slot IDs in all communication cycles are assigned to this node.

The modification of the assignment of key slots shall only be possible during the *POC:config* state. The node shall ignore attempts to change the assignment of a key slot in all other states.

### 13.3.2.4 Communication slot assignment for reception

A node in the *POC:normal active* state can receive all frames and therefore all messages on the FlexRay bus. Frames which are relevant for the operation of the protocol (sync frames, for example) will be processed by the protocol engine automatically without the need of explicit assignment of the corresponding communication slots for reception.

This specification gives no guidance on how communication slot assignment for reception shall be implemented.

See 13.3.2.8.2, which describes requirements for message buffers used for reception.

#### 13.3.2.5 Conflicting communication slot assignment for reception and transmission

In the static segment if a node has been configured to receive a message in a specific slot it shall only do so if this slot has not also been assigned to the node for transmission. Transmission of messages in the static segment always has precedence over reception.

In the dynamic segment transmission has precedence over reception in a specific slot if there is at least one transmit message buffer with valid payload data configured for this specific slot.

#### 13.3.2.6 Non-queued message buffers

##### 13.3.2.6.1 Structure and general behaviour

Message transmission and reception operate on message buffers. A non-queued message buffer is a structure which consists of

— message buffer configuration data,

— message buffer status data, and

— message data.

Upon a transition from either the *POC:normal active* or *POC:normal passive* state to either the *POC:halt* or *POC:ready* state the CHI shall continue to provide the host access to the data that it would have provided had the POC remained in the *POC:normal active* or *POC:normal passive* states. Note that at this point the CHI will no longer update the data as the protocol engine will stop providing new data.

The behaviour of the CHI upon attempted host access to buffer status or payload data that has never been updated by the CHI because no data was provided by the protocol engine is implementation dependent. It is required that the access to such data does not give the appearance that data was received or transmitted when such a reception or transmission did not actually take place.

##### 13.3.2.6.2 Message buffer configuration data

The message buffer configuration data shall contain at least the following parameters for each message buffer:

— a type indication (transmit or receive message buffer);

— the channel identifier (A, B, or, for dual channel devices, A&B) on which the node shall transmit or receive a message using this buffer;

— the slot identifier of the communication slot in which the node shall transmit or receive a message using this buffer;

— the set of communication cycles in which the node shall transmit or receive a message using data provided in this message buffer.

Additionally a message buffer configured for transmission needs to contain the following configuration parameters:

— the length of the available message data in the buffer (*MessageLength*)[141];

141) In the static segment *MessageLength* describes the number of two-byte words that are provided to the protocol engine by the CHI. The macro ASSEMBLE_STATIC_FRAME assures that a frame with the correct length (i.e., *gPayloadLengthStatic*) is transmitted by the protocol engine. In the dynamic segment this value describes the actual

— the payload preamble indicator (for the network management service or message ID filtering);

— the header CRC;

— a transmission mode indicator (single shot mode or continuous mode).

The previously identified buffer configuration information is divided into two classes.

— Class 1 Buffer Configuration Data:

  — the type indication;

  — the channel identifier;

  — the slot identifier;

  — the set of communication cycles.

— Class 2 Buffer Configuration Data.

  — the length of the available message data;

  — the payload preamble indicator;

  — the header CRC;

  — the transmission mode indicator.

The host shall have read access to both classes of message buffer configuration data independently of the protocol state.

The host shall have write access to both classes of message buffer configuration data in the *POC:config* state.

An implementation shall allow configuration of message buffers in all states other than *POC:default config*, and shall prevent configuration of message buffers in the *POC:default config* state. An implementation shall provide a configurable mechanism to prevent the host from writing certain classes of message buffer configuration data in states other than *POC:config*. The configuration of this mechanism, i.e., the selection of which classes of message buffer configuration data can be written, shall only be possible in the *POC:config* state. At a minimum, an implementation shall support the following configurations.

— No write access to either Class 1 or Class 2 message buffer configuration data while in states other than *POC:config* and *POC:default config.*

— Write access to both Class 1 and Class 2 message buffer configuration data while in states other than *POC:config* and *POC:default config.*

— Write access to Class 2 message buffer configuration data but no write access to Class 1 buffer configuration data while in states other than *POC:config* and *POC:default config.*

An implementation shall be capable of applying the previous configurations to all non-queued message buffers[142)] but an implementation may provide more fine-grained control, allowing sets of message buffers to have different restrictions.



---

number of two-byte words in the frame to be transmitted.

142) An exception to this requirement applying to all buffers is that it is allowable that message buffers exclusively associated with key slots are always restricted from configuration in states other than *POC:config*.

The implementation shall ensure that it is not possible for the host to change the configuration data of a message buffer while the buffer is being used according to 13.3.2.8.3 or while it is updated by the controller.

#### 13.3.2.6.3 Message buffer status data

The message buffer status shall be able to provide the following information to the host:

— slot status;

— a slot status updated indicator;

— channel indication of the channel which the slot status refers to.

The message buffer status shall be able to indicate the status information listed in 13.3.2.8.4 for a transmit message buffer or the status information listed in 13.3.2.9.3.1 for a receive message buffer.

The CHI shall indicate that the slot status information of a message buffer has been updated by setting the corresponding slot status updated indicator to true.

When the host (re)configures a message buffer (i.e., when the host performs a write access to the class 1 or class 2 configuration data for a buffer), the CHI shall set the slot status updated indicator of the corresponding message buffer to false.

The CHI shall set the slot status updated indicator to false for all message buffers upon a transition from *POC:ready* state to either *POC:coldstart listen*, *POC:external startup* or *POC:integration listen* (i.e., upon a transition of the variable *vPOC!State* from READY to STARTUP).

#### 13.3.2.6.4 Message buffer payload data and payload data valid flag

The payload data to be transmitted in a frame or the payload data which a node receives will be stored in the message buffer data portion of the corresponding message buffer (see subclauses 13.3.2.9.2 and 13.3.2.8.2).

The CHI will grant the following access rights to the message buffer payload data:

— for transmit buffers the host shall have read and write access to the data;

— for receive buffers the host shall have read access to the data.

The CHI will write the data on reception of the corresponding frame when the data is provided by the protocol engine.

The CHI shall ensure that it always provides a consistent set of configuration and status and message data to the host in case of receive message buffers and to the protocol engine in case of transmit message buffers.

Each message buffer shall provide a Boolean payload data valid flag associated with the message buffer payload data.

The host shall be granted the same access rights to this flag as for the message buffer payload data.

By setting this flag to true in a transmit message buffer the host indicates that the message buffer payload data in the message buffer is ready for transmission, i.e. the payload data is "valid". The CHI can change the value of the payload data valid flag when the corresponding frame has been sent depending on the transmission mode indicator (see 13.3.2.8.2).

For receive buffers the CHI sets the payload data valid flag to true when the message buffer payload data actually contains the payload data of a received frame.

When the host (re)configures a message buffer (i.e., when the host performs a write access to the class 1 or class 2 configuration data for a buffer), the CHI shall set the payload data valid flag of the corresponding message buffer to false.

The CHI shall set the payload data valid flag to false for all receive message buffers upon a transition from *POC:ready* state to either *POC:coldstart listen*, *POC:external startup* or *POC:integration listen* (i.e., upon a transition of the variable *vPOC!State* from READY to STARTUP). The payload data valid flag for transmit message buffers shall not be modified upon a transition from POC:ready state to any state[143) 144)].

The CHI shall set the payload data valid flag to false for all transmit message buffers upon a transition into the *POC:ready* state from any state other than *POC:config*, *POC:wakeup listen*, *POC:wakeup send*, and *POC:wakeup detect* (i.e., when the *vPOC!State* variable changes from either STARTUP, NORMAL_ACTIVE, or NORMAL_PASSIVE to READY)[145)].

#### 13.3.2.6.5 Buffer enabling and buffer locking

For the purposes of the selection of buffers, this part of ISO 17458-2 defines two additional concepts related to the availability of buffers – an enabled / disabled status and a locked / unlocked (buffer locking) status.

In this context, a buffer being "enabled" refers to whether or not a buffer's configuration (see 13.3.2.6.2) is complete. A buffer that has not been configured, or is in the process of being configured or reconfigured, is considered disabled. A buffer whose configuration is complete, and is not in the process of being reconfigured, is considered enabled.

NOTE The enabled / disabled status of a buffer has nothing to do with the host reading or making modifications to the status or payload data of the buffer (see buffer locking, described below).

Buffer locking refers to an optional, implementation-specific mechanism often used to ensure that the host's accesses to a buffer's slot status and payload data information are atomic. When a buffer is locked the CHI will not update the buffer's slot status or payload data information, and will inform the protocol engine (via the *vTCHI!TxMessageAvailable* flag) that no data is available from the buffer. Buffer locking is an optional capability of an implementation, but if present, the locked / unlocked status shall be considered when deciding if a buffer is a candidate (see 13.3.2.7.2 and 13.3.2.7.3).

### 13.3.2.7 Non-queued message buffer identification

#### 13.3.2.7.1 Principles of message buffer selection

In case that the host has configured several non-queued message buffers for transmission and / or reception in a specific slot the CHI has to select an "active message buffer" at the beginning of this specific slot in a deterministic way according to the following general steps which will be detailed in the following subclauses.

a) Identify the set of "candidate transmit buffers" and the set of "candidate receive buffers", i.e. message buffers which have been configured for the specific slot where the transmission or reception shall happen.

b) Identify one buffer out of the set of the candidate transmit buffers for the specific channel as the "selected transmit buffer" and identify it as the "active message buffer" for the channel. If no transmit buffer is found then identify one buffer out of the set of the candidate receive buffers for the specific channel as the "selected receive buffer" and identify it as the "active message buffer" for the channel.

143) This allows an application to pre-configure payload data for transmission in the *POC:ready* or *POC:config* states, i.e. to prepare buffers for transmission before issuing the RUN command.
144) The system designer should be aware that after issuing a RUN command the payload data valid flag is only cleared for receive buffers - the payload data valid flag for transmit buffers remains in the original state.
145) This prevents unintentional transmissions due to payload data valid flags left over from previous operation (for example, if an IMMEDIATE_READY command is followed by a RUN command).

These steps shall be performed on a per channel basis, i.e., if the communication controller is configured for dual channel operation the identification process shall be performed for each channel without considering the result of the other channel. As a result the CHI in this case may identify two active message buffers, one for each channel[146)].

#### 13.3.2.7.2 Candidate transmit message buffer identification

If a slot is assigned to the node according to the transmission slot assignment list the CHI shall evaluate

— the message buffer configuration (type indication, slot identifier, channel identifier and set of communication cycles) and the payload data valid flag of all non-queued message buffers AND

— the current status of the buffer locking (if applicable) AND

— the segment in which the slot with the specific slot identifier is located AND

— the current protocol state *vPOC!State* of the protocol engine AND

— the current slot mode *vPOC!SlotMode* of the protocol engine

to identify the set of the candidate transmit message buffers for this slot.

The following table determines for a given buffer whether it becomes a member of the set of candidate transmit buffers (value true in the "candidate" column).

The table uses the term "buffer match" to indicate that

— the type indication of the buffer identifies it as transmit buffer AND

— the slot identifier matches the slot number of the specific slot AND

— the channel identifier is included in *pChannels* AND

— the specific communication cycle is in the set of communication cycles AND

— the message buffer is enabled (see 13.3.2.6.5) by the host AND

— if the implementation supports buffer locking (see 13.3.2.6.5) the message buffer is not locked by the host.

Buffer match evaluates to true if all the conditions above are met. It evaluates to false if one or more of these conditions are not met.

Table 8 specifies the transmit message buffer candidate.



146) A message buffer that is configured for both channels can be the only active message buffer.

**Table 8 — Transmit message buffer candidate**

| Buffer match | Segment | vPOC!State | payload data valid flag | vPOC!SlotMode | Candidate |
|---|---|---|---|---|---|
| true | dynamic | NORMAL_ACTIVE | true | KEYSLOT or ALL_PENDING | false |
| true | dynamic | NORMAL_ACTIVE | true | ALL | true |
| true | dynamic | NORMAL_ACTIVE | false | don't care | false |
| true | dynamic | all but NORMAL_ACTIVE | don't care | don't care | false |
| true | static | don't care | don't care | don't care | true |
| false | don't care | don't care | don't care | don't care | false |

If a slot is not assigned to the node for transmission according to the transmission slot assignment list, then the set of the candidate transmit message buffers for this slot is empty.

#### 13.3.2.7.3 Candidate receive message buffer identification

A non-queued message buffer will be included in the set of candidate receive message buffers for a slot when

— the type indication of the buffer is set to receive buffer AND

— the slot identifier matches the slot number of the specific slot AND

— the channel identifier is included in *pChannels* AND

— the specific communication cycle is in the set of communication cycles AND

— the message buffer is enabled (see 13.3.2.6.5) by the host AND

— if the implementation supports buffer locking (see 13.3.2.6.5), the message buffer is not locked by the host AND

— for the static segment only the CHI does not consider the slot / channel combination to be assigned for transmission (i.e., the value of *vTCHI!Assignment* that is passed to the protocol engine is UNASSIGNED).

If none of the configured receive message buffers fulfils all these conditions, then the set of candidate receive message buffers is empty for the slot.

#### 13.3.2.7.4 Selected transmit buffer identification

In case that more than one message buffer is member of the set of candidate transmit message buffers, the CHI has to select one specific message buffer out of this set. This selection process is implementation dependent, but has to meet all of the following requirements.

— The selection process has to be deterministic so that the host can predict the result.

— If there is at least one candidate transmit message buffer there shall be a selected transmit message buffer.

— The selected transmit message buffer shall be one of the candidate transmit message buffers.

— If one or more of the transmit candidate message buffers have the payload data valid flag set to true the selected message buffer shall be one of these.

— If the set of candidate transmit message buffers is empty, no buffer will become the selected transmit buffer.

#### 13.3.2.7.5 Selected receive buffer identification

In case that more than one message buffer is member of the set of candidate receive message buffers, the CHI has to select one specific message buffer out of this set. This selection process is implementation dependent, but has to meet all of the following requirements.

— The selection process has to be deterministic so that the host can predict the result.

— If there is at least one candidate receive message buffer there shall be a selected receive message buffer.

— The selected receive message buffer shall be one of the candidate receive message buffers.

— If the set of candidate receive message buffers is empty, no buffer will become the selected receive buffer.

#### 13.3.2.7.6 Active message buffer identification

The active message buffer is determined according to the following rules.

— If there is a selected transmit message buffer then the active message buffer is the selected transmit message buffer.

— If there is no selected transmit message buffer but there is a selected receive message buffer the active message buffer is the selected receive message buffer.

— If there is neither a selected transmit message buffer nor a selected receive message buffer then there is no active message buffer.

Depending on the value of the type indication the active message buffer will be called "active transmit message buffer" or "active receive message buffer" in the following subclauses.

For a given channel there can be at most one active message buffer at a time - independent of its type.

### 13.3.2.8 Message transmission

#### 13.3.2.8.1 General concept

Message transmission is primarily determined by the concept of the transmission slot assignment list (see 13.3.2.3). Transmit message buffers provide the information (e.g. payload data) which shall be transmitted but in principle transmission could take place without a transmit buffer being configured.

#### 13.3.2.8.2 Transmit buffer configuration

A transmit buffer shall be configured for a transmission based on the channel identifier and the slot assignment information describing in which slot the transmission shall occur.

For frames transmitted in the static segment, the following channel configuration shall be supported:

— configured for channel A;

— configured for channel B;

— for dual channel devices, configured for channel A and for channel B.

For frames transmitted in the dynamic segment, the following channel configurations shall be supported:

— configured for channel A;

— configured for channel B.

It shall be possible to configure a transmit buffer for a single slot or for a set of slots sharing the same slot identifier in a configurable set of communication cycles.

The configuration of the set of communication cycles shall support, at a minimum, the ability to define sets of communication cycles which equal the sets which can be defined using a Cycle_Repetition and a Cycle_Offset to determine the configuration:

— The transmit buffer is configured for a transmission slot if *vCycleCounter* mod Cycle_Repetition = Cycle_Offset

with

— Cycle_Repetition selected from the set of {1, 2, 4, 5, 8, 10, 16, 20, 32, 40, 50, 64};

— Cycle_Offset selected from the set {0 … 63} with Cycle_Offset < Cycle_Repetition.

In case the CHI allows that multiple buffers are configured for the same slot the CHI has to select a unique buffer for transmission in a deterministic way which is predictable by the host (see 13.3.2.6.5).

Each transmit buffer is associated with a payload data valid flag and a transmission mode indicator.

The payload data valid flag denotes whether the message contained in the transmit buffer is valid or not.

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

— is provided with a consistent set of valid message data from the transmit buffer, or

— receives an indication that a consistent read of message data is not possible or that the transmit buffer contains no valid message (i.e., when the payload data valid flag is set to false).

The CHI shall support at least two modes, which determine how the payload data valid flag shall be updated after a transmission.

— Single shot transmission mode:
When payload data has been provided by the host and marked as valid by setting the payload data valid flag to true, the data remains valid until the data has been transmitted (i.e., the protocol engine returns *vSS!FrameSent* as true). After the transmission, the CHI shall automatically invalidate the data by setting the payload data valid flag to false (i.e., the payload data is transmitted exactly once[147)] as the result of the buffer update by the host).

— Continuous transmission mode:
When payload data has been provided by the host and marked as valid by setting the payload data valid flag to true, the data remains valid until the host explicitly marks the data as invalid by setting the payload data valid flag to false (i.e., the payload data is transmitted repeatedly until the host invalidates the buffer data).

147) In the static segment null frames may be transmitted when the payload data valid flag is false.

#### 13.3.2.8.3 Transmit buffer identification for message retrieval

The protocol engine interacts with the controller host interface by importing the message data provided by the CHI at the start of each slot according to the media access control processes defined in clause 9. The protocol engine imports data elements from the CHI as defined in 9.3.

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

a) The CHI shall check whether the slot is assigned to the node on the relevant channel by querying the transmission slot assignment list for the relevant channel using *vCycleCounter* and the channel-specific value for *vSlotCounter*.

b) If the slot is not assigned to the node then the CHI shall return *vTCHI* with *vTCHI!Assignment* set to UNASSIGNED else *vTCHI!Assignment* shall be set to ASSIGNED and the subsequent steps shall be executed.

c) *vTCHI!HeaderCRC* shall be set to the value of the header CRC retrieved from the transmission slot assignment list[148)].

d) The active transmit message buffer shall be identified according to the process described in 13.3.2.7.

e) If there is no active transmit message buffer then the CHI shall signal to the protocol engine that the communication slot is assigned but without any message data available by setting *vTCHI!TxMessageAvailable* to false and returning *vTCHI*.

f) If there is an active transmit message buffer then a consistent read of its data shall be attempted.

g) If a consistent read is not possible (i.e., if the buffer is locked[149)]), or the payload data valid flag is set to false, then the CHI shall signal to the protocol engine that the communication slot is assigned but without any message data available by setting *vTCHI!TxMessageAvailable* to false and returning *vTCHI*.

h) If a consistent read is possible, the CHI shall signal to the protocol engine that the message data is available for this communication slot by setting

   — *vTCHI!TxMessageAvailable* to true,

   — *vTCHI!PPIndicator* to the value retrieved from the transmit buffer configuration data as defined by the message ID filtering service in 13.3.3.3 and the network management service in 13.3.3.4,

   — *vTCHI!Length* to the length of the message *MessageLength* held in the corresponding transmit buffer,

   — *vTCHI!Message* to the message data from the transmit buffer

   and returning *vTCHI*.

#### 13.3.2.8.4 Transmit buffer status

A message buffer configured for transmission shall be able to hold a snapshot of the following status information:

148) The header CRC information in the transmission slot assignment list may be derived dynamically from information in the transmit buffer configuration.

149) See 13.3.2.6.5. In most circumstances a buffer could not be an active transmit message buffer if it is locked. It is possible, however, that the buffer is locked by the host after it has already been selected as the active transmit message buffer.

— a frame transmitted indicator that shall be set to true if a frame that was not a null frame was completely transmitted[150]. The CHI shall set the frame transmitted indicator to true if there was a complete frame transmission in the slot (*vSS!FrameSent* is true), and shall leave the frame transmitted indicator at its current value if there was not a complete frame transmission in the slot (*vSS!FrameSent* is false). The CHI shall provide a mechanism which allows the host to reset the frame transmitted indicator;

— 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);

— 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);

— 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);

— 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);

— a valid frame flag that shall reflect the status of the *vSS!ValidFrame* variable in the transmission slot status[151].

The slot status updated indicator shall be set to true by the CHI when the slot status has been updated. The CHI shall provide a mechanism that allows the host to reset this indicator.

If a transmit buffer is configured for both channel A and channel B it shall be capable of storing the above listed status information separately for each channel, and it shall be possible to determine the corresponding channel for each set of status information.

#### 13.3.2.9 Message reception

##### 13.3.2.9.1 Receive buffer types

Message reception operates on queued and / or non-queued receive buffers. A non-queued receive buffer is a data storage structure

— for which the host has access to the data through a read operation,

— for which the protocol engine has access to the data through a write operation, and

— in which new values overwrite former values.

Refer to subclauses 13.3.2.9.2 and 13.3.2.9.3 for the requirements on non-queued receive buffers.

A queued receive buffer is a data storage structure

— for which the host has access to the data through a read operation,

150) A frame is considered completely transmitted at the start of transmission of the FES. See the *frame transmitted on A* signal in Figure 55 for details.

151) In most cases *vSS!ValidFrame* will be set to false if a transmit buffer was selected as the active buffer, but there are circumstances where it could be set to true even though a transmit buffer was selected (if, for example, transmission is prohibited because the MAC is in the KEYSLOTONLY mode or if the node is in the *POC:normal passive* state).

— for which the protocol engine has access to the data through a write operation, and

— in which new values are queued behind former values.

Refer to 13.3.2.11 for the requirements on queued receive buffers / FIFO's.

#### 13.3.2.9.2 Non-queued receive buffer configuration

For each slot on each channel the protocol engine provides a tuple of values to the CHI consisting of a slot status *vSS* of the communication slot in the current communication cycle and, if a semantically valid frame was received in this communication slot on this channel, the contents *vRF* of the first semantically valid frame. Subclause 13.3.2.7 describes how a specific receive buffer is selected based on this tuple.

In general, a receive buffer will be configured with the following:

— a slot identifier configuration identifying a single slot identifier;

— a channel configuration identifying a set of channels (A, B, or both A and B);

— a cycle counter configuration identifying a set of communication cycles.

It shall be possible to configure a receive buffer for a single slot or for a set of slots sharing the same slot identifier in a configurable set of communication cycles.

The configuration of the set of communication cycles shall support, at a minimum, defining sets of communication cycles which equal the sets which can be defined using a Cycle_Repetition and a Cycle_Offset to determine the configuration:

The receive buffer is configured for a reception slot if

— *vCycleCounter* mod Cycle_Repetition = Cycle_Offset

with

— Cycle_Repetition selected from the set of {1, 2, 4, 5, 8, 10, 16, 20, 32, 40, 50, 64}

— Cycle_Offset selected from the set {0 … 63} with Cycle_Offset < Cycle_Repetition

For frames received in the static segment, the following channel configuration shall be supported:

— receive buffer configured for channel A;

— receive buffer configured for channel B;

— for dual channel devices, receive buffer configured for both channel A and channel B. In this case the CHI shall select among the information provided by the channel specific FSP processes of the protocol engine. If the protocol engine provides only a single valid frame, that frame should be stored in the buffer, regardless of which channel it was received on. If the protocol engine provides two valid frames (one from each channel), and only one of the frames is a non-null frame, the non-null frame should be stored in the buffer, regardless of which channel this non-null frame was received on. If the protocol engine provides two valid frames and both are non-null frames[152)] the receive buffer shall store the frame that was received on channel A[153)].

152) If both valid frames are null frames only the slot status from both channels is stored in the buffer.

153) The preference for channel A is entirely arbitrary, serving only to define a deterministic behaviour.

Subclause 13.3.2.7 defines the requirements for resolving situations when more than one non-queued receive buffer can serve as a candidate buffer for a given slot / cycle / channel combination. This process can result in ambiguity in a case where one buffer is configured for channel A only (or channel B only, or channel A&B) while another buffer is configured for both channel A and channel B for the same (or an overlapping) slot / cycle combination. This specification makes no requirements on the buffer selection behaviour of an implementation for such configurations – the behaviour in these circumstances is implementation dependent[154)].

For frames received in the dynamic segment, the following channel configurations shall be supported:

— receive buffer configured for channel A;

— receive buffer configured for channel B.

For each receive buffer the CHI shall ensure that the host either

— is provided with a consistent set of message data from the receive buffer, or

— receives an indication that a consistent read of the message data is not possible.

For each receive buffer the CHI shall ensure that the information provided by the protocol engine is written to the corresponding receive buffer either

— consistently, i.e. perform a consistent write of its data as if in one indivisible operation, or

— not at all. In this case the payload data valid flag shall be set to false, so the host can assess that receive buffer contents were lost.

In case the CHI allows that multiple buffers are configured for the same slot the CHI has to select a unique buffer in a deterministic way which is predictable by the host (see 13.3.2.6.5).

If for the same slot both a receive buffer and a transmit buffer are configured, by following the process described in 13.3.2.6.5 the CHI will ensure that the content of the active transmit buffer is provided to the protocol engine. In such a case the transmission has priority over the reception.

Each receive buffer shall hold up to a buffer specific bound of two-byte words.

For non-queued receive buffers this buffer specific bound may be set individually for each receive buffer within a node between 1 and *cPayloadLengthMax*.

#### 13.3.2.9.3 Non-queued receive buffer contents

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

##### 13.3.2.9.3.1 Slot status data

A message buffer configured for reception shall be able to store a snapshot of the following slot status variables:

— a valid frame flag that shall be set if a syntactically and semantically correct frame was received in the corresponding slot (*vSS!ValidFrame* set to true) and cleared if no valid frame was received (*vSS!ValidFrame* set to false);

154) The use of such a configuration is not recommended.

— a syntax error flag that shall be set if a syntax error was observed in the corresponding slot (*vSS!SyntaxError* set to true) or cleared if no syntax error was observed in the corresponding slot (*vSS!SyntaxError* set to false);

— a content error flag that shall be set if a content error was observed in the corresponding slot (*vSS!ContentError* set to true) or cleared if no content error was observed in the corresponding slot (*vSS!ContentError* set to false);

— a slot boundary violation flag that shall be set if a slot boundary violation, i.e. channel active at the start or at the end of the slot, was observed in the corresponding slot (*vSS!BViolation* set to true) or cleared if no slot boundary violation was observed in the corresponding slot (*vSS!BViolation* set to false);

— a null frame indicator flag that shall be set according to the value of the *vSS!NFIndicator* status of the corresponding slot. If no valid frame was received the flag *vSS!NFIndicator* will be set to 0. This flag will only have a value of 1 if the slot contained a valid, non-null frame.

The slot status updated indicator shall be set to true for the message buffer by the CHI when the slot status has been updated. The CHI shall provide a mechanism that allows the host to reset this indicator.

If a receive buffer is configured for both channel A and channel B it has to be capable of storing the above listed status information separately for each channel, and it shall be possible to determine the corresponding channel for each set of status information.

Table 9 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[155)] and semantically[156)] valid frames received in a static or dynamic slot, respectively.



155) A frame is syntactically valid if it fulfils the decoding rules defined in 7.3.5 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.

156) A semantically valid frame is a syntactically valid frame that also fulfils a set of content related criteria.

**Table 9 — Slot status interpretation**

| Valid Frame | Syntax Error | Content Error | BViolation | 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 |

[a] The syntax error indication may be caused by additional activity, but it could also be caused by a decoding error in the second bit of the FES of a frame otherwise free of decoding errors. In the latter case, there may or may not be additional activity.

#### 13.3.2.9.3.2 Frame contents data

A message buffer configured for reception shall be able to store a snapshot of the following frame contents variables at the end of the communication slot if a valid frame was received in the slot (*vSS!ValidFrame* is equal to true) and the frame contained valid payload data (*vRF!Header!NFIndicator* is equal to one)[157)] and the buffer was the active message buffer:

— the reserved bit *vRF!Header!Reserved*;

— the frame ID *vRF!Header!FrameID*;

— the cycle counter *vRF!Header!CycleCount*;

— the length field *vRF!Header!Length*;

— the header CRC *vRF!Header!HeaderCRC*;

— the payload preamble indicator *vRF!Header!PPIndicator*;

157) The reception of a null frame should not cause a snapshot of the frame contents data to be stored in the buffer.

— a flag that indicates that the buffer has been updated at some point during operation. This flag should be set to zero at buffer configuration, and be set to one when the buffer is updated as a result of the reception of a valid, non-null frame. Note that this behaviour could be achieved by simply copying the *vRF!Header!NFIndicator* flag whenever the *vRF* structure is passed to the CHI from the protocol engine;

— the sync frame indicator *vRF!Header!SyFIndicator*;

— the startup frame indicator *vRF!Header!SuFIndicator*;

— *vRF!Header!Length* number of two-byte payload data words from *vRF!Payload*, if *vRF!Header!Length* does not exceed the buffer length *BufferLength* of the receive message buffer;

— *BufferLength* number of two-byte payload data words from *vRF!Payload*, if *vRF!Header!Length* exceeds the buffer length *BufferLength* of the receive message buffer[158)];

— if a receive buffer is configured for both channel A and channel B it shall also store the channel indicator *vRF!Channel* allowing the determination of the source of the frame contents data stored in the buffer.

An exception to the normally required behaviour exists during the first slot of the first cycle of operation of a TT-E coldstart node. Due to the short time between availability of information on the current cycle count and the start of the first cycle of operation it may not be possible for an implementation to complete a search of the entire set of buffers during the first slot. As a result, it may be possible that even though a buffer may be configured that an implementation may not be aware of this in time to behave as described in this part of ISO 17458. As a result, a specific exception is made for the first slot of the first cycle after a TT-E coldstart node's transition from the *POC:external startup* state to the *POC:normal active* state - such a node is allowed, but not required, to update the frame contents data for a receive buffer for this slot.

#### 13.3.2.10 Non-queued message buffer status update

For a given channel the protocol engine makes the slot status *vSS* available to the CHI at the end of each slot[159)].

In case that there is an active message buffer available for the given channel the CHI shall copy a snapshot of the relevant *vSS* status information into the status data of the active message buffer (see subclauses 13.3.2.8.4 and 13.3.2.9.3.1).

The active message buffer is available for the status update when

— the active message buffer has been identified at the start of the slot

AND

— the host did not reconfigure the active message buffer between the time of the identification of the active message buffer and the point in time the status update should occur

AND

— the active message buffer is not locked at the time the status update should occur (see 13.3.2.6.5)

AND in case of an active transmit message buffer

— there has been no write access by the host to the message buffer between the time the payload is provided to the protocol engine and the point in time at which the status update should occur.

158) The host can assess such a truncation through the data element *vRF!Header!Length*.
159) During the startup phase no buffer update will take place.

If an active message buffer is configured for both channels, only the portion of the status relevant for the channel(s) for which the active message buffer has been identified will be updated[160)].

In case there is no active message buffer available for the given channel at the time of the slot status update no non-queued message buffer will be updated with status information.

An exception to the normally required behaviour exists during the first slot of the first cycle of operation of a TT-E coldstart node. Due to the short time between availability of information on the current cycle count and the start of the first cycle of operation it may not be possible for an implementation to complete a search of the entire set of buffers during the first slot. As a result, it may be possible that even though a buffer may be configured that an implementation may not be aware of this in time to behave as described in this part of ISO 17458. As a result, a specific exception is made for the first slot of the first cycle after a TT-E coldstart node's transition from the *POC:external startup* state to the *POC:normal active* state - such a node is allowed, but not required, to update the frame contents data for a receive buffer for this slot.

#### 13.3.2.11 General concept

##### 13.3.2.11.1 The concept of queued receive buffers

Queued receive buffers, also referred to as FIFO buffers, are a class of receive buffer that are capable of storing status information and payload data for more than one frame. Information is placed into the FIFO by the CHI and is removed from the FIFO by the host. The FIFO represents a queue, i.e., it is possible that multiple messages go into the FIFO before any of the messages are removed by the host, and it is possible that the host reads multiple messages that have already been previously placed in the FIFO even though no additional messages have been placed into the FIFO.

The FlexRay FIFO preserves the order of the messages which are placed into the FIFO - messages are removed from the FIFO in the same order they are placed into the FIFO (i.e., it has a "First In First Out" behaviour). Refer to 13.3.2.11.4 for additional details.

When the host removes a message from the FIFO the removal frees up FIFO resources that can be used to store additional messages in the FIFO. As long as the host removes the entries from the FIFO often enough that the FIFO does not fill up, all messages that should go into the FIFO are made available to the host[161)].

##### 13.3.2.11.2 Basic FIFO behaviour

###### 13.3.2.11.2.1 Design of a FIFO buffer

A FIFO buffer consists of a number of entries, each of which is capable of storing information related to the reception of a frame. Status variables and frame contents data are only stored in a FIFO buffer when the admittance criteria are passed (see 13.3.2.11.3) and the FIFO buffer is selected (see 13.3.2.11.2.2).

Unlike non-queued receive buffers, FIFO buffers have no requirement to store slot status information for a slot if no valid frame was received.



160) Under certain configurations this could result in only one of the two sets of channel-specific status information being updated. This could happen, for example, if the configuration allows a buffer configured for both channels to be selected as the active buffer on one channel and not the active buffer on the other channel. Since there is only one slot status updated flag for a buffer, the host would not be able to determine that such a "half update" took place. As a result, configurations in which this could occur should be avoided.

161) If the host does not remove the messages from the FIFO often enough eventually the FIFO will reach its limits and an overrun will occur. The behaviour of the FIFO in this case is implementation dependent.

A FIFO buffer shall be able to store a snapshot of the following slot status variables when a frame is admitted into the FIFO:

— a syntax error flag that shall be set if a syntax error was observed in the corresponding slot (*vSS!SyntaxError* set to true) or cleared if no syntax error was observed in the corresponding slot (*vSS!SyntaxError* set to false);

— a content error flag that shall be set if a content error was observed in the corresponding slot (*vSS!ContentError* set to true) or cleared if no content error was observed in the corresponding slot (*vSS!ContentError* set to false);

— a slot boundary violation flag that shall be set if a slot boundary violation, i.e. channel active at the start or at the end of the slot, was observed in the corresponding slot (*vSS!BViolation* set to true) or cleared if no slot boundary violation was observed in the corresponding slot (*vSS!BViolation* set to false);

— a null frame indicator flag that shall be set according to the value of the *vSS!NFIndicator* status of the corresponding slot. If no valid frame was received the flag *vSS!NFIndicator* will be set to 0. This flag will only have a value of 1 if the slot contained a valid, non-null frame.

A FIFO buffer shall be able to store a snapshot of the following frame contents variables when a frame is admitted into the FIFO:

— the reserved bit vRF!Header!Reserved;

— the frame ID vRF!Header!FrameID;

— the cycle counter vRF!Header!CycleCount;

— the length field vRF!Header!Length;

— the header CRC vRF!Header!HeaderCRC;

— the payload preamble indicator vRF!Header!PPIndicator;

— The sync frame indicator vRF!Header!SyFIndicator;

— the startup frame indicator vRF!Header!SuFIndicator;

— vRF!Header!Length number of two-byte payload data words from vRF!Payload, if vRF!Header!Length does not exceed the width of the selected FIFO buffer (see 13.3.2.11.4);

— a number of two-byte words equal to the width of the selected FIFO buffer from vRF!Payload, if vRF!Header!Length exceeds the width of the selected FIFO buffer;

— if a FIFO may be configured to admit frames from both channel A and channel B it shall be capable of storing the channel indicator vRF!Channel allowing the determination of the source of the frame contents data stored in a FIFO entry.

Unlike non-queued receive buffers, FIFO buffers have no requirement to store slot status information for a slot if no valid frame was received.

For each FIFO buffer entry the CHI shall ensure that the host either

— is provided with a consistent set of message data from the receive buffer, or

— receives an indication that a consistent read of the message data is not possible.

For each FIFO buffer entry the CHI shall ensure that the data provided by the protocol engine is written to the corresponding FIFO buffer either

— consistently, i.e. perform a consistent write of its data in one indivisible operation, or

— not at all. In this case a flag shall be provided through which the host can assess that receive buffer contents were lost.

#### 13.3.2.11.2.2 Admittance into a FIFO

A FIFO buffer has a set of admittance criteria (that determines when the CHI puts frame status and payload data from the protocol engine into the FIFO). When a frame matches all of the admittance criteria (see 13.3.2.11.3) it is placed into the FIFO.

A frame is only considered for admittance into a FIFO if no match for the frame is found within the configured non-queued receive buffers (i.e., the FIFO's have lower priority than the non-queued receive buffers described in 13.3.2.9.2).

If the CHI supports multiple FIFO's and if the admittance criteria for the FIFO's are configurable such that a single frame can meet the admittance criteria of more than one FIFO the CHI shall select a unique FIFO buffer in a deterministic, implementation dependent manner.

The host shall have read access to admittance criteria independent of the protocol state.

The host shall have write access to the admittance criteria in the *POC:config* state.

Enabling and disabling the write access to the admittance criteria during *POC:normal active* state and *POC:normal passive* state shall only be possible in the *POC:config* state.

Host write access to the admittance criteria shall be possible in *POC:normal active* state and in *POC:normal passive* state, if this was explicitly enabled during *POC:config* state.

When the host is changing the set of admittance criteria in the *POC:normal active* state or in the *POC:normal passive* state, the CHI shall prevent that an inconsistent set of admittance criteria (i.e., the changes are incomplete) is applied.

After changing the admittance criteria in the *POC:normal active* state or *POC:normal passive* state, the CHI shall continue to provide host access to the existing entries in the FIFO. New entries shall be placed into the FIFO according to the new admittance criteria.

An exception to the normally required behaviour exists during the first slot of the first cycle of operation of a TT-E coldstart node. Due to the short time between availability of information on the current cycle count and the start of the first cycle of operation it may not be possible for an implementation to complete a search of the entire set of non-queued buffers during the first slot. As a result, it may be possible that even though a non-queued buffer is configured that an implementation may not be aware of this in time to behave as described in this part of ISO 17458. Further, it may not be possible for an implementation that implements the majority of the FIFO admittance checks prior to the start of a slot to complete all checks in time to process a frame received in the first slot of operation. As a result, a specific exception is made for the first slot of the first cycle after a TT-E coldstart node's transition from the *POC:external startup* state to the *POC:normal active* state - such a node is allowed, but not required, to consider a received frame for admittance into a FIFO buffer. If an implementation does consider a frame for admission it shall meet all of the requirements described in this subclause, specifically, that there be no match for the frame within the configured non-queued receive buffers [162)].



162) As a result, an implementation that is not able to search all non-queued buffers is not allowed to consider the frame for admittance into a FIFO buffer.

#### 13.3.2.11.2.3 Reading and removal from a FIFO

The CHI shall provide a method to allow the host to read the slot status data and frame contents data stored in the first (oldest) FIFO entry. In addition, the CHI shall provide a mechanism to allow the host to remove the first entry from the FIFO without requiring the host to read the entire FIFO entry.

It shall be possible for the host to read any portion of the slot status data and / or the frame contents data of the first entry in the FIFO and then make a decision as to whether or not to read the remaining data in the FIFO entry and still have access to the remaining data, i.e., it is not acceptable for a read of any portion of the data alone to cause data to be lost[163)].

It shall be possible for the host to read information out of the FIFO (and remove messages from the FIFO) at that same time that other frames are being placed into the FIFO.

Upon a transition from either the *POC:normal active* or *POC:normal passive* state to either the *POC:halt* or *POC:ready* state the CHI shall continue to provide host access to the FIFO buffer entries that it would have provided had the POC remained in the *POC:normal active* or *POC:normal passive* states.

The behaviour of the CHI upon attempted host access to queued buffer status or payload data that has never been updated is implementation dependent. It is required, however, that the access to such data does not give the appearance that data was received when such a reception did not actually take place.

### 13.3.2.11.3 FIFO admittance criteria

#### 13.3.2.11.3.1 Overview

An implementation shall be capable of determining which frames will be placed into a FIFO structure. The decision as to whether or not a frame is placed into a FIFO buffer is based on a series of five admittance criteria:

— FIFO frame validity admittance criteria;

— FIFO channel admittance criteria;

— FIFO frame identifier admittance criteria;

— FIFO cycle counter admittance criteria;

— FIFO message identifier admittance criteria.

Each received frame for which there is no active non-queued buffer is a candidate for admission into a FIFO received buffer[164)]. Each candidate frame is checked against the FIFO admittance criteria to determine if is placed into a FIFO receive buffer.

A frame shall pass all five of the admittance criteria in order to be placed into a FIFO buffer - if one or more of the admittance criteria fail the frame will not be placed into a FIFO buffer.

An implementation shall support the admittance criteria (and configuration of the admittance criteria) as described in the following subclauses.

---

163) In this context being "lost" is different from being removed from the FIFO - depending on the implementation, it may be possible to remove an entry from the FIFO without the data being lost.

164) An interesting situation can arise if a non-queued buffer is configured to receive a frame on both channels, and valid frames actually occur on both channels. In this case, even though the payload information for only one of the frames can be stored in the receive buffer (see 13.3.2.9.2), both frames actually have an active non-queued buffer (for example, the active buffer will store slot status information affected by the frames on both channels). As a result, neither frame would be a candidate for admission into a FIFO receive buffer.

#### 13.3.2.11.3.2 FIFO frame validity admittance criteria

An implementation shall be capable of admitting or not admitting a frame into the FIFO based on the validity of the frame. In addition, an implementation shall be capable of being configured to admit or not admit valid null frames into the FIFO. Specifically, the frame validity admittance criteria shall have a single Boolean configuration, AdmitNullFrame.

The frame validity admittance criteria shall have the following behaviour.

— If AdmitNullFrame = false,
then all valid non-null frames (i.e., frames that cause an FSP process to generate an 'update *vRF* on A in CHI' or 'update *vRF* on B in CHI' - refer to Figure 118) shall be considered to pass the FIFO frame validity admittance criteria. All other frames (or other activity) shall be considered to fail the frame validity admittance criteria.

— If AdmitNullFrame = true,
then all valid frames (*vSS!ValidFrame* is equal to true) shall be considered to pass the FIFO frame validity admittance criteria[165)]. All other frames (or other activity) shall be considered to fail the frame validity admittance criteria.

#### 13.3.2.11.3.3 FIFO channel admittance criteria

An implementation shall be capable of being configured to admit or not admit a frame into the FIFO based on the channel on which the frame was received. Specifically, an implementation shall be capable of configuring the FIFO channel admittance criteria such that a frame shall be considered to pass these admittance criteria

— only if the frame was received on Channel A, or

— only if the frame was received on Channel B, or

— regardless of whether the frame was received on Channel A or Channel B[166)].

If the FIFO channel admittance criteria is configured for the last option (i.e., to admit frames regardless of channel), and for a given slot frames are received on both channel A and channel B, and both frames pass all of the other admittance criteria, then both frames shall be accepted into a FIFO structure[167)].

#### 13.3.2.11.3.4 FIFO frame identifier admittance criteria

An implementation shall be capable of being configured to admit or not admit a frame into the FIFO based on the Frame ID[168)] of the received frame. Specifically, the FIFO frame identifier admittance criteria shall be considered to be passed only if the received frame identifier belongs to a configurable set of frame identifiers referred to as the FIFO frame identifier set. If the received frame identifier is not a member of the FIFO frame identifier set the FIFO frame identifier admittance criteria shall be considered to be failed.

---

165) These criteria include all frames that would be admitted when AdmitNullFrame = false, but also include valid null frames.

166) It is required that dual channel implementations be able to be configured to support the reception of frames on either channel A or channel B into some FIFO structure, but if more than one FIFO structure is available, then it is not necessary that frames from channel A and channel B be received into the same FIFO structure. See 13.3.2.11.4 for further information.

167) The required behaviour of the FIFO differs from the optional behaviour of receive buffers in the static segment defined in 13.3.2.9.2. If the admittance criteria is set for both channel A and channel B, then both frames, rather than just the first valid non-null frame, shall be entered into a FIFO structure.

168) This admittance criterion is actually based on the current value of the slot counter at the time the frame is received rather than on the Frame ID received in the frame. Since the frame validity admittance criteria will only admit valid frames into the FIFO, the only frames that could be admitted in the FIFO are those whose received frame identifier matches the current slot counter. The term frame identifier is used in this subclause for clarity only - strictly speaking, the implementation would be based on the slot counter at the time of frame reception.

At a minimum, the FIFO frame identifier set shall be capable of being configured to any set that could be configured with the following abstract definition[169)].

Define four parameters, Range1min, Range1max, Range2min, and Range2max, each configurable in the range of {0 … *cSlotIDMax*}. The FIFO frame identifier set is the set of all frame identifiers Frame ID in the range {1 … *cSlotIDMax*} such that

— $Range1_{min}$ <= Frame ID <= $Range1_{max}$

or

— $Range2_{min}$ <= Frame ID <= $Range2_{max}$

NOTE The individual ranges identified above can cross the boundary between the static and dynamic segment, i.e., Range1min could be an identifier within the static segment and Range1max could be an identifier within the dynamic segment. The FIFO frame identifier definition above applies even if Frame ID lies in a different segment than the minimum or maximum value of the range configuration.

#### 13.3.2.11.3.5 FIFO cycle counter admittance criteria

An implementation shall be capable of being configured to admit or not admit a frame into the FIFO based on the cycle counter value at the time the frame is received. Specifically, the FIFO cycle counter admittance criteria shall be considered to be passed only if the cycle counter value when the frame was received belongs to a configurable set of cycle counter values referred to as the FIFO cycle counter set. If the cycle counter when the frame was received is not a member of the FIFO cycle counter value set the FIFO cycle counter value admittance criteria shall be considered to be failed.

At a minimum, the FIFO cycle counter set shall be capable of being configured to any set that could be configured with the following abstract definition[170)].

Define two parameters, Cycle_Repetition and Cycle_Offset, with:

— Cycle_Repetition selected from the set of {1, 2, 4, 5, 8, 10, 16, 20, 32, 40, 50, 64};

— Cycle_Offset selected from the set {0 .. 63};

— Cycle_Offset < Cycle_Repetition;

The FIFO cycle counter value set is the set of all cycle counter values Cycle_Counter in the range of {0 .. 63} such that *vCycleCounter* mod Cycle_Repetition = Cycle_Offset

#### 13.3.2.11.3.6 Message identifier admittance criteria

An implementation shall be capable of being configured to admit or not admit a frame into the FIFO based on the value of the optional Message ID that can be present in frames received in the dynamic segment.

Message identifier admittance criteria shall have the following characteristics.

169) An implementation does not need to explicitly support the parameters in the abstract definition, but needs to be able to generate all of the FIFO frame identifier sets that could be generated by the abstract definition.

170) An implementation does not need to explicitly support the parameters in the abstract definition, but needs to be able to generate all of the FIFO cycle counter sets that could be generated by the abstract definition.

The message identifier admittance criteria shall have three configurable parameters[171]:

— MsgIDMask, an integer in the range of {0 … 65 535}, which determines the bitwise AND mask value for the admittance criteria;

— MsgIDMatch, an integer in the range of {0 … 65 535}, which determines the bitwise comparison value for the admittance criteria;

— AdmitWithoutMessageID, a Boolean configuration which determines whether or not frames received in the dynamic segment that don't contain a message ID will be admitted into the FIFO.

The message identifier admittance criteria shall have the following behaviour:

— If the frame was received in the static segment the frame is considered to have passed the message identifier admittance criteria, or

— If the frame was received in the dynamic segment and does not contain a message identifier (i.e., the payload preamble indicator of the frame is set to zero), the behaviour depends on the AdmitWithoutMessageID configuration of the FIFO.

  — If AdmitWithoutMessageID = false the frame is considered to fail the message identifier admittance criteria.

  — If AdmitWithoutMessageID = true the frame is considered to pass the message identifier admittance criteria.

or

— If the frame was received in the dynamic segment and does contain a message identifier (i.e., the payload preamble indicator of the frame is set to one), the admittance depends on the value of the message identifier, MessageID, and the FIFO configurations MsgIDMask and MsgIDMatch. Specifically, the frame is considered to pass the message identifier admittance criteria if

```
( MessageID & MsgIDMask ) = MsgIDMatch
```

otherwise the frame is considered to fail the message identifier admittance criteria. Here the "&" symbol represents a bitwise AND of the binary representation of the values of MessageID and MsgIDMask[172].

#### 13.3.2.11.4 FIFO performance requirements

An implementation shall provide at least one FIFO receive buffer structure.

A FIFO receive buffer structure shall have a depth of at least eight entries (i.e., in the absence of entries being removed from the FIFO by the host, the FIFO shall be capable of storing at least eight frames from the protocol engine without any loss or overwrite of frame contents data or slot status data).

Each entry in a FIFO receive structure also has a "width", i.e., each entry in a FIFO shall be capable of storing a number of bytes of frame payload data greater than or equal to the implementation's capability to store payload data for non-queued receive buffers. If a valid message is received whose payload is longer than the

171) Unlike other FIFO admittance criteria, an implementation shall explicitly support the specified parameters of the message ID admittance criteria, i.e., it is not acceptable to support other means of configuring these admittance criteria.

172) For example, if MsgIDMask = 3 855 (0x0F0F) and MsgIDMatch = 1 537 (0x0601), then a frame with a MessageID of 38 641 (0x96F1) would pass the admittance criteria, but a frame with MessageID 1 538 (0x0602) would fail the criteria. Note that a configuration with MsgIDMask = 0 and MsgIDMatch = 0 would pass this criteria for any frame regardless of message identifier.

configured width of a FIFO entry, or is longer than the implementation's maximum width for FIFO's, the implementation shall store the message's first payload bytes up to the configured or maximum length in the FIFO entry.

At a minimum an implementation shall provide the capability to store at least eight entries with the width defined above (i.e., the depth and width requirements shall be met simultaneously).

An implementation that supports two channels shall be capable of receiving frames from each channel into some FIFO structure[173)].

The requirements on buffer depth and width, FIFO admittance criteria, etc. are channel independent (i.e., the FIFO's of an implementation need to be able to support the indicated number of frames entirely on channel A, entirely on channel B, or any mixture of channel A and B[174)].

NOTE there is no requirement that an implementation needs to support the indicated number of messages on both channel A and Channel B at the same time, or that the FIFO admittance criteria are able to be independently set for Channel A and Channel B.

A dual channel implementation shall be able to support simultaneous reception on channel A and B (i.e., if frames appear on both channels at the same time, or with arbitrary overlap, and both frames meet the admittance criteria for the FIFO, then both shall be entered into some FIFO receive buffer structure).

An implementation shall place frames into a FIFO in the order that they were received (i.e., a frame that was received earlier would be removed from a FIFO before a frame that was received later). There is no requirement for an implementation to maintain the relative order of frames received into different FIFO's, but it is required that a FIFO preserve the relative order of all messages received into the same FIFO.

#### 13.3.2.11.5 FIFO status information

The following information on the status of a FIFO receive buffer structure shall be provided in the CHI.

— The number of occupied entries currently in the FIFO

— An overrun indicator is set if a FIFO overrun condition has occurred. An overrun occurs when a frame matches all of the FIFO admittance criteria but the FIFO is not capable of increasing the total number of entries in the FIFO. The overrun indicator shall remain set until explicitly cleared by the host. The behaviour of the FIFO upon the occurrence of an overflow condition, and in recovery from an overflow condition, is implementation dependent.

— Information that allows the host to determine how much of the FIFO remains available to accept additional information. The form of this information depends on the structure of the FIFO and is implementation dependent[175)].

In addition to the previous status information, an implementation shall provide the ability to notify the host via an interrupt request when the available resources of the FIFO have fallen below a configurable level.

The nature of the configurability is dependent on the structure of the FIFO and is implementation dependent[176)].

---

173) The intention is to allow implementations that dedicate separate FIFO's for each channel as well as implementations that support a single FIFO that can receive messages from either channel.

174) For example, a dual channel implementation shall be able support the reception of eight frames from channel A alone, or eight frames from channel B alone, seven from channel A and one from channel B, six from channel A and two from channel B, or any combination that adds up to eight frames total.

175) For fixed size FIFO's with a fixed number of entries the number of messages currently in the FIFO would meet this requirement. For FIFO's with variable size structures some other implementation dependent mechanism to determine the amount of the FIFO resources available shall be provided.

176) For example, a FIFO that offers a fixed number of fixed size entries might be configurable based on the number of

### 13.3.3 CHI Services

#### 13.3.3.1 Macrotick timer service

The CHI shall provide at least two absolute timers capable of notifying the host at expiration.

Each of the required timers shall, at a minimum, be capable of being configured with the following expiration criteria.

— To expire at an absolute time in terms of cycle count and macrotick, i.e. the timer would expire at a configurable macrotick in a configurable communication cycle, and

— To expire at a configurable macrotick only (i.e., the timer would expire at a configurable macrotick independent of cycle count)

In addition, each of the required timers shall, at a minimum, be capable of supporting the following two modes of operation.

— A non-repetitive mode of operation where the timer will expire once the configured expiration criteria occurs and will not expire again until restarted or reconfigured by the host, and

— A repetitive mode of operation where the timer will expire every time the configured expiration criteria occurs (i.e., the timer can expire multiple times without further interaction from the host).

It shall be possible to configure and activate a timer when the protocol is in either the *POC:normal active* state or the *POC:normal passive* state[177)]. All absolute timers shall be deactivated when the protocol leaves the *POC:normal active* state or the *POC:normal passive* state apart from transitions between the *POC:normal active* state and the *POC:normal passive* state. It shall also be possible for the host to deactivate an absolute timer in any state that allows the timer to be activated. Once deactivated, a timer shall be explicitly activated again by the host before it can expire again.

#### 13.3.3.2 Interrupt service

The interrupt service provides a set of configurable interrupt requests to the host based on a set of interrupt sources reflecting events that occur in the protocol engine or the CHI.

At a minimum, an implementation shall provide the following interrupt sources.

— Each timer (as described in 13.3.3.1) provided by an implementation shall be able to act as an interrupt source with the event being the expiration of the timer.

— The FlexRay cycle shall act as an interrupt source with the event being the start of a FlexRay cycle. This requirement is in addition to the previous timer requirement (i.e., it is not acceptable to meet this requirement through the use of a timer as specified in 13.3.3.1).

— State transitions of the Protocol Operation Control process shall act as an interrupt source. This source shall only signal an event whenever the POC transitions to the *POC:halt*, *POC:ready*, *POC:normal active* or *POC:normal passive* states for any reason other than the processing of an IMMEDIATE_READY or FREEZE command.

— Each FIFO structure shall act as an interrupt source with the event being a frame reception that causes the available resources of the FIFO structure fall below a configurable level (refer to 13.3.2.11.5).

---

entries in the FIFO. A FIFO based on variable size entries might be configurable based on the fraction of the storage space remaining in the FIFO.

177) It is allowed, but not required, for an implementation to support configuration/activation of an absolute timer in states other than *POC:normal active* and *POC:normal passive*.

An implementation shall provide a mechanism that allows the events of interrupt sources to generate interrupt requests (i.e., maps interrupt source events to interrupt requests). In general, a single interrupt request is allowed to support more than one interrupt source, however at least one of the mandated absolute timers (see 13.3.3.1) shall provide a dedicated interrupt request that is not shared with any other interrupt source.

It shall be possible for the host to enable and disable the generation of interrupt requests for interrupt sources. When interrupt request generation is enabled for an interrupt source the event defined for the interrupt source shall cause the generation of a corresponding interrupt request. When interrupt request generation is disabled for an interrupt source the event defined for the interrupt source shall not cause the generation of a corresponding interrupt request.

An implementation shall provide control of interrupt request generation with at least two levels. At the first level, it shall be possible for the host to individually control whether interrupt request generation is enabled or disabled for each interrupt source. At the second level, it shall be possible for the host to globally disable all interrupt request generation regardless of the individual interrupt request generation enabled / disabled status of each interrupt source. Specifically, if interrupt request generation is globally disabled no interrupt source will generate interrupt requests; if interrupt request generation is not globally disabled each interrupt source shall generate interrupt requests according to its individual interrupt request generation status.

An implementation shall provide an interrupt status indication for each interrupt source. The interrupt status indication for an interrupt source shall be set when the interrupt source's event occurs[178)] and shall remain set until reset under control of the host.

#### 13.3.3.3 Message ID filtering service

The message ID filtering service provides means for selecting receive buffers based on a message ID that may be exchanged in the first two bytes of the payload segment of selected frames within the dynamic segment that have the payload preamble indicator set to one in the header of the frame.

To support this service the message buffer configuration data shall allow the host to configure the payload preamble indicator for each transmit buffer so that the host can configure whether the message data contains a message ID or not (refer to 13.3.2.6.2).

Message ID filtering is required for at least one FIFO receive buffer structure as described in subclauses 13.3.2.11.3.6 and 13.3.2.11.4.

#### 13.3.3.4 Network management service

The network management service provides means for exchanging and processing network management data. This service supports high-level host-based network management protocols that provide cluster-wide coordination of shutdown decisions based on the actual application state. The network management service may also be used by applications to implement other functionality.

Network management is performed by exchanging a network management vector in selected network management enabled frames within the static segment of the communication cycle that have the payload preamble indicator set to one in the header of the frame. The payload preamble indicator of the message buffer configuration data allows the host to configure whether or not a message contains a network management vector.

Throughout each communication cycle the CHI shall maintain an accrued network management vector by applying a bit-wise OR between the current accrued network management vector and each[179)] network

178) The interrupt status indication of an interrupt source is set whenever the corresponding event occurs irrespective of whether interrupt request generation is individually enabled for the interrupt source or whether interrupt request generation is globally disabled.

179) Only valid frames reported to the CHI (via the *vRF* structures of the channel-specific FSP processes) are considered. If more than one frame occurs in a slot on a given channel only the first valid frame (i.e., the one reported in *vRF*) is considered.

management vector received in a valid frame on each channel (regardless of whether or not any receive buffer is configured to explicitly receive the valid frame)[180].

— If *pNMVectorEarlyUpdate* is set to false, the protocol status data shall contain a snapshot of the accrued network management vector that shall be updated no sooner than the end of the cycle and no later than the availability of the NIT status information or the availability of the payload data from the last static slot, whichever occurs later. The snapshot shall contain the value of the accrued network management vector at the end of the cycle, and shall include the effects of any reception that may have occurred in the last static slot.

— If *pNMVectorEarlyUpdate* is set to true, the protocol status data shall contain a snapshot of the accrued network management vector that shall be updated no sooner than the end of the static segment and no later than the availability of the payload data from the last static slot. The snapshot shall contain the value of the accrued network management vector at the end of the static segment (i.e., shall include the effects of any reception that may have occurred in the last static slot).

— These updates take place as long as the protocol is in either the *POC:normal active* state or the *POC:normal passive* state.

— The accrued network management vector is set to zero at the beginning of each communication cycle[181].

— Following the completion of startup, the NM vector snapshot shall be set to all zeros prior to the first update of the NM vector snapshot (when this occurs depends on the configuration parameter *pNMVectorEarlyUpdate*).

In addition to the capabilities provided above, it is also possible for implementers to provide additional types of Network Management Services, for example, providing direct (non-OR'd) access to the NM Vector data.



180) In this context, a frame is only considered valid if *vSS!ValidFrame* status is set to true when the protocol engine exports the slot status *vSS* to the CHI at the end of a slot or segment. For example, frames that are received in a slot which is also used for transmission shall not be accrued into the network management vector since such frames will have *vSS!ValidFrame* set to false when the slot status is exported to the CHI.

181) Even though in some circumstances the snapshot of the NM vector might be presented to the CHI after the end of the cycle, it shall represent the status of the accrued NM vector from the previous cycle. As a result, the snapshot shall be taken before the vector is set to zero at the beginning of the subsequent cycle.

# Annex A
(normative)

# System parameters

## A.1 Protocol constants

Table A.1 defines the Protocol constants.

**Table A.1 — Protocol constants**

| Name | Description | Value |
|---|---|---|
| *cChannelIdleDelimiter* | Duration of the channel idle delimiter. | 11 gdBit |
| *cClockDeviationMax* | Maximum clock frequency deviation, equivalent to 1500 ppm (1500 ppm = 1500 / 1000000 = 0,0015). | 0,0015 |
| *cCrcInit[A]* | Initialisation vector for the calculation of the frame CRC on channel A (hexadecimal). | 0xFEDCBA |
| *cCrcInit[B]* | Initialisation vector for the calculation of the frame CRC on channel B (hexadecimal). | 0xABCDEF |
| *cCrcPolynomial* | Frame CRC polynomial (hexadecimal). | 0x5D6DCB |
| *cCrcSize* | Size of the frame CRC calculation register. | 24 bits |
| *cCycleCountMax* | Maximum cycle counter value in any cluster. | 63 |
| *cdBSS* | Duration of the Byte Start Sequence. | 2 gdBit |
| *cdCAS* | Duration of the logical low portion of the collision avoidance symbol (CAS) and media access test symbol (MTS). | 30 gdBit |
| *cdCASActionPointOffset* | Initialisation value of the CAS action point offset timer. | 1 MT |
| *cdCASRxLowMin* | Lower limit of the CAS acceptance window. | 29 gdBit |
| *cdCycleMax* | Maximum cycle length. | 16 000 µs |
| *cdCycleStartTimeout* | Maximum allowed jitter between the 'external cycle start' from the time gateway source and the internal 'cycle start' of the time gateway sink. | 5 µT |
| *cdFES* | Duration of the Frame End Sequence. | 2 gdBit |
| *cdFSS* | Duration of the Frame Start Sequence. | 1 gdBit |
| *cdInternalRxDelayMax* | Maximum value of the implementation specific delay on the receive path of the decoder. | 4 samples [a] |
| *cdInternalRxDelayMin* | Minimum value of the implementation specific delay on the receive path of the decoder. | 1 sample [a] |

[a] This value is based on the experience of the semiconductor manufacturers.

**Table A.1 —** *(continued)*

| Name | Description | Value |
|---|---|---|
| *cdMaxMTNom* [b] | Maximum duration of a nominal macrotick. An implementation shall be able to support nominal macrotick durations between *cdMinMTNom* and *cdMaxMTNom*. | 6 µs |
| *cdMinMTNom* [c] | Minimum duration of a nominal macrotick. An implementation shall be able to support nominal macrotick durations between *cdMinMTNom* and *cdMaxMTNom*. | 1 µs |
| *cdStaggerDelay* | Delay used to stagger the deactivation of the TxD and TxEN outputs during CAS / MTS and WUP transmission to eliminate the possibility of brief glitches. The TxD output will remain LOW for *cdStaggerDelay* following the deactivation of TxEN. | 1 gdBit |
| *cdTSrcCycleOffset* | The delay between the cycle starts of the time gateway source and time gateway sink for a cluster operating in TT-E external sync mode. | 40 µT |
| *cdWakeupMaxCollision* | Number of continuous bit times at LOW during the idle phase of a WUS that will cause a sending node to detect a wakeup collision. | 5 gdBit |
| *cdWakeupTxActive* | Duration of the LOW phase of a transmitted wakeup symbol and the active (i.e., HIGH or LOW) phases of a transmitted WUDOP. | 6 µs |
| *cdWakeupTxIdle* | Duration of the idle phase between two low phases inside a wakeup pattern. | 18 µs |
| *cFrameThreshold* | Threshold used to differentiate noise from activity arising from a frame in the dynamic segment media access. Activity exceeding this threshold is assumed to have come from frame transmission as opposed to noise. | 80 gdBit |
| *cHCrcInit* | Initialisation vector for the calculation of the header CRC on channel A or channel B (hexadecimal). | 0x01A |
| *cHCrcPolynomial* | Header CRC polynomial (hexadecimal). | 0x385 |
| *cHCrcSize* | Size of header CRC calculation register. | 11 bits |
| *cMicroPerMacroMin* | Minimum number of microticks per macrotick during the offset correction phase. | 20 µT |
| *cMicroPerMacroNomMin* | Minimum number of microticks in a nominal (uncorrected) macrotick. | 40 µT |
| *cMicroPerMacroNomMax* | Maximum number of microticks in a nominal (uncorrected) macrotick. | 240 µT |
| *cPayloadLengthMax* | Maximum length of the payload segment of a frame. | 127 two-byte words |
| *cPropagationDelayMax* | Maximum allowable propagation delay arising from the physical layer and analogue effects inherent in the FlexRay CC's involved in the transmission and reception of a communication element. These are the delays that occur between the points labelled as TP1_FF and TP4_FF in Figure 110 of ISO 17458-4. | 2,5 µs |
| *cSamplesPerBit* | Number of samples taken in the determination of a bit value. | 8 |
| *cSlotIDMax* | Highest slot ID number. | 2 047 |
| *cStaticSlotIDMax* | Highest static slot ID number. | 1 023 |

b This parameter is only introduced to be able to define a minimum conformance class range that all implementations shall support. Note that this macrotick duration may not be achievable for all microtick durations - see B.4.5 for details.

c This parameter is only introduced to be able to define a minimum conformance class range that all implementations shall support. Note that this macrotick duration may not be achievable for all bit rates or microtick durations - see B.2 and B.4.5 for details.

**Table A.1 —** *(continued)*

| Name | Description | Value |
|---|---|---|
| *cStrobeOffset* | Sample where bit strobing is performed (first sample of a bit is considered as sample 1). | 5 |
| *cSyncFrameIDCountMax* | Maximum number of distinct sync frame identifiers that may be present in any cluster. | 15 |
| *cVotingDelay* | Number of samples of delay between the RxD input and the majority voted output in the glitch-free case. | (*cVoting-Samples* -1) / 2 |
| *cVotingSamples* | Numbers of samples in the voting window used for majority voting of the RxD input. | 5 |

## A.2 Performance constants

Table A.2 defines the performance constants.

**Table A.2 — Performance constants**

| Name | Description | Value |
|---|---|---|
| *cdMaxOffsetCalculation* | Maximum time allowed for calculation of the offset correction value, measured from the end of the static segment. In some situations the offset correction calculation deadline is actually longer - see 12.6.2 for details. | 1 350 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 12.6.3 for details. | 1 500 µT |

# Annex B
(normative)

# Configuration constraints

## B.1 General

This appendix specifies the configurable parameters of the FlexRay protocol. This appendix also identifies the configurable range of the parameters, and gives constraints on the values that the parameters may take on. All implementations that support a given parameter shall support at least the parameter range identified in this appendix. An implementation is allowed, however, to support a broader range of configuration values.

Following functions are used for the configuration parameter calculation:

— Function **ceil(**x**)** returns the nearest integer greater than or equal to x;

— Function **floor(**x**)** returns the nearest integer less than or equal to x;

— Function **max(**x1; x2;…; xn**)** returns the maximum value from the set of arguments {x1, x2,…, xn }. If the arguments xi are compound expressions composed of multiple parameters, then the values selected for each of the parameters should be the ones that maximize the overall value of xi;

— Function **min(**x1; x2;…; xn**)** returns the minimum value from the set of arguments {x1, x2,…, xn }. If the arguments xi are compound expressions composed of multiple parameters, then the values selected for each of the parameters should be the ones that minimize the overall value of xi;

— Function **round(**x**)** returns the integer value closest to x using asymmetric arithmetic rounding;

— Function **if(**c; x; y**)** returns x if condition c is true, otherwise y;

— [ ] denotes units;

— $\mathbf{max}_{M,N}\mathbf{(...)}$ means the maximum of all paths from node M to node N with M,N = 1, ..., number of nodes and M <> N;

— Function **or(**c1; c2**)** returns true if either condition c1 or condition c2 (or both) are true, otherwise returns false;

— Function $\mathbf{min}_N(x_N)$ returns the value x of node N that represents the minimum of all nodes of a cluster;

— Function $\mathbf{max}_N(x_N)$ returns the value x of node N that represents the maximum of all nodes of a cluster.

## B.2 Bit rates

The FlexRay data link layer specification defines three standard bit rates – 10 Mbit / s, 5 Mbit / s, and 2,5 Mbit / s. The configuration ranges shown in this appendix reflect the necessary parameter ranges for an implementation that supports operation at all three standard speeds.

## B.3 Parameters

### B.3.1 Global cluster parameters

#### B.3.1.1 Protocol relevant

Protocol relevant global cluster parameters are parameters used within the SDL models to describe the FlexRay protocol. They shall have the same value in all nodes of a cluster.

Table B.1 defines the Global protocol relevant parameters.

**Table B.1 — Global protocol relevant parameters**

| Name | Description | Range |
|---|---|---|
| *gColdstartAttempts* | Maximum number of times a node in the cluster is permitted to attempt to start the cluster by initiating schedule synchronisation. | 2 – 31 |
| *gCycleCountMax* [a] | Maximum cycle counter value in a given cluster. | [7, 9 , ..., *cCycleCountMax*] |
| *gdActionPointOffset* | Number of macroticks the action point is offset from the beginning of a static slot. | 1 – 63 MT |
| *gdCASRxLowMax* | Upper limit of the CAS acceptance window. | 28 – 254 gdBit |
| *gdDynamicSlotIdlePhase* | Duration of the idle phase within a dynamic slot. | 0 – 2 Minislot |
| *gdIgnoreAfterTx* | Duration that bit strobing is paused after a transmission. | 0 – 15 gdBit |
| *gdMinislot* | Duration of a minislot. | 2 – 63 MT |
| *gdMinislotActionPointOffset* | Number of macroticks the minislot action point is offset from the beginning of a minislot. | 1 – 31 MT |
| *gdStaticSlot* | Duration of a static slot. | 3 – 664 MT |
| *gdSymbolWindow* | Duration of the symbol window. | 0 – 162 MT |
| *gdSymbolWindowAction-PointOffset* | Number of macroticks the action point is offset from the beginning of the symbol window. | 1 – 63 MT |
| *gdTSSTransmitter* | Number of bits in the Transmission Start Sequence. | 1 – 15 gdBit |
| *gdWakeupRxIdle* | Number of bits used by the node to test the duration of the 'idle' or HIGH phase of a received wakeup. | 8 – 59 gdBit |
| *gdWakeupRxLow* | Number of bits used by the node to test the duration of the LOW phase of a received wakeup. | 8 – 59 gdBit |
| *gdWakeupRxWindow* | The size of the window, expressed in bits, used to detect wakeups. | 76 – 485 gdBit |
| *gdWakeupTxActive* | Number of bits used by the node to transmit the LOW phase of a wakeup symbol and the HIGH and LOW phases of a WUDOP. | 15 – 60 gdBit |
| *gdWakeupTxIdle* | Number of bits used by the node to transmit the 'idle' part of a wakeup symbol. | 45 – 180 gdBit |

[a] This parameter shall be an odd integer.

**Table B.1 —** *(continued)*

| Name | Description | Range |
|---|---|---|
| *gListenNoise* | Upper limit for the startup listen timeout and wakeup listen timeout in the presence of noise. This is used as a multiplier of the node parameter *pdListenTimeout*. | 2 – 16 |
| *gMacroPerCycle* | Number of macroticks in a communication cycle. | 8 – 16 000 MT |
| *gMaxWithoutClockCorrectionFatal* | Threshold used for testing the *vClockCorrectionFailed* counter. Defines the number of consecutive even / odd cycle pairs with missing clock correction terms that will cause the protocol to transition from the *POC:normal active* or *POC:normal passive* state into the *POC:halt* state.[b] | 1 – 15 even / odd cycle pairs |
| *gMaxWithoutClockCorrectionPassive* | Threshold used for testing the *vClockCorrectionFailed* counter. Defines the number of consecutive even / odd cycle pairs with missing clock correction terms that will cause the protocol to transition from the *POC:normal active* state to the *POC:normal passive* state.[b] | 1 – 15 even / odd cycle pairs |
| *gNumberOfMinislots* | Number of minislots in the dynamic segment. | 0 – 7 988 |
| *gNumberOfStaticSlots* | Number of static slots in the static segment. | 2 – *cStaticSlotIDMax* |
| *gPayloadLengthStatic* | Payload length of a static frame.[c] | 0 – *cPayloadLengthMax* two-byte words |
| *gSyncFrameIDCountMax* | Maximum number of distinct sync frame identifiers present in a given cluster. | 2 – *cSyncFrameIDCountMax* |

b If *gMaxWithoutClockCorrectionPassive* is set to a value greater than or equal to *gMaxWithoutClockCorrectionFatal* then the CC enters the *POC:halt* state directly (i.e., without first entering the *POC:normal passive* state).

c All static frames in a cluster have the same payload length. For 2,5 Mbit / s the payload length is restricted by the maximum transmission duration of *adTxMax*. See B.4.41.

### B.3.1.2 Protocol related

Protocol related global cluster parameters are parameters that have a meaning in the context of the FlexRay protocol but are not used within the SDL models. These parameters are used in the configuration constraints. They shall have the same value in all nodes of a cluster.

Table B.2 defines the global protocol related parameters.

**Table B.2 — Global protocol related parameters**

| Name | Description | Range |
|---|---|---|
| *gChannels* | The channels that are used by the cluster. | [A, B, A&B] |
| *gClockDeviationMax* | Maximum frequency deviation of the time sources inside a cluster from their nominal frequencies. | 0 < *gClockDeviationMax* <= *cClockDeviationMax* |
| *gClusterDriftDamping* | The cluster drift damping factor, based on the longest microtick *adMicrotickMax* used in the cluster. Used to compute the local cluster drift damping factor *pClusterDriftDamping*. | 0 – 5 µT |
| *gdBit* | Nominal bit time. | [0,1, 0,2, 0,4] µs |
| *gdCycle* [a] | Length of the cycle. | 24 µs – *cdCycleMax* |
| *gdMacrotick* | Duration of the cluster wide nominal macrotick. | 1 – 6 µs |
| *gdNIT* | Duration of the Network Idle Time. | 2 – 15 978 MT |
| *gdSampleClockPeriod* | Sample clock period. | [0,0125, 0,025, 0,05] µs |
| *gExternOffsetCorrection* | External offset correction value applied in a cluster. | 0 – 0,35 µs |
| *gExternRateCorrection* | External rate correction value applied in a cluster. | 0 – 0,35 µs |
| *gNetworkManagement-VectorLength* | Length of the Network Management vector in a cluster. | 0 – 12 bytes |

a See the calculation for the minimum value of *pMicroPerCycle* in B.4.16. Maximum value is given by *cdCycleMax*. The minimum value is a theoretical minimum. Implementations may require larger minimum cycle length because of other conditions (for example performance constants given in Table A.2).

## B.3.2 Node parameters

### B.3.2.1 Protocol relevant

Protocol relevant node parameters are parameters used within the SDL models to describe the FlexRay protocol. They may have different values in different nodes of a cluster.

Table B.3 defines the local node protocol relevant parameters.

**Table B.3 — Local node protocol relevant parameters**

| Name | Description | Range |
|---|---|---|
| *pAllowHaltDueToClock* | Boolean parameter that controls the transition to the *POC:halt* state due to clock synchronisation errors.<br>If set to true, the CC is allowed to transition to *POC:halt*.<br>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 shall have valid clock correction terms before the CC will be allowed to transition from the *POC:normal passive* state to *POC:normal active* state. If set to zero, the CC is not allowed to transition from *POC:normal passive* to *POC:normal active*. | 0 – 31 even / odd cycle pairs |
| *pChannels* | Channels to which the node is connected. | [A, B, A&B] |
| *pClusterDriftDamping* | Local cluster drift damping factor used for rate correction. | 0 – 10 µT |
| *pdAcceptedStartupRange* | Expanded range of measured clock deviation allowed for startup frames during integration. | 29 – 2 743 µT |
| *pDecodingCorrection* | Value used by the receiver to calculate the difference between primary time reference point and secondary time reference point. | 12 – 136 µT |
| *pDelayCompensation[A]*, *pDelayCompensation[B]* | Value used to compensate for reception delays on the indicated channel. | 4 – 211 µT |
| *pdListenTimeout* | Value for the startup listen timeout and wakeup listen timeout. Although this is a node local parameter, the real time equivalent of this value should be the same for all nodes in the cluster. | 1 926 – 2 567 692 µT |
| *pExternalSync* | Parameter indicating whether the node is externally synchronized (operating as time gateway sink in an TT-E cluster) or locally synchronized. If *pExternalSync* is set to true then *pTwoKeySlotMode* shall also be set to true. | Boolean |
| *pExternOffsetCorrection* | Number of microticks added or subtracted to the NIT to carry out a host-controlled external offset correction. | 0 – 28 µT |
| *pExternRateCorrection* | Number of microticks added or subtracted to the cycle to carry out a host-controlled external rate correction. | 0 – 28 µT |
| *pFallBackInternal* | Parameter indicating whether a time gateway sink node will switch to local clock operation when synchronisation with the time gateway source node is lost (*pFallBackInternal* = true) or will instead go to *POC:halt* (*pFallBackInternal* = false). | Boolean |
| *pKeySlotID* | ID of the key slot, i.e., the slot used to transmit the startup frame, sync frame, or designated key slot frame. If this parameter is set to zero the node does not have a key slot. | 0 – *cStaticSlotIDMax* |
| *pKeySlotOnlyEnabled* | Parameter indicating whether or not the node shall enter key slot only mode following startup. | Boolean |

**Table B.3 —** *(continued)*

| Name | Description | Range |
|---|---|---|
| *pKeySlotUsedForStartup* | Parameter indicating whether the key slot(s) are used to transmit startup frames. If *pKeySlotUsedForStartup* is set to true then *pKeySlotUsedForSync* shall also be set to true. If *pTwoKeySlotMode* is set to true then both *pKeySlotUsedForSync* and *pKeySlotUsedForStartup* shall also be set to true. | Boolean |
| *pKeySlotUsedForSync* | Parameter indicating whether the key slot(s) are used to transmit sync frames. If *pKeySlotUsedForStartup* is set to true then *pKeySlotUsedForSync* shall also be set to true. If *pTwoKeySlotMode* is set to true then both *pKeySlotUsedForSync* and *pKeySlotUsedForStartup* shall also be set to true. | Boolean |
| *pLatestTx* | Number of the last minislot in which a frame transmission can start in the dynamic segment. | 0 – 7 988 Minislot |
| *pMacroInitialOffset[A],*<br>*pMacroInitialOffset[B]* | Integer number of macroticks between the static slot boundary and the following macrotick boundary of the secondary time reference point based on the nominal macrotick duration. | 2 – 68 MT |
| *pMicroInitialOffset[A],*<br>*pMicroInitialOffset[B]* | Number of microticks between the secondary time reference point and the macrotick boundary immediately following the secondary time reference point. The parameter depends on *pDelayCompensation[Ch]* and therefore it has to be set independently for each channel. | 0 – 239 µT |
| *pMicroPerCycle* | Nominal number of microticks in the communication cycle of the local node. If nodes have different microtick durations this number will differ from node to node. | 960 – 1 280 000 µT |
| *pOffsetCorrectionOut* | Magnitude of the maximum permissible offset correction value. | 15 – 16 082 µT |
| *pOffsetCorrectionStart* | Start of the offset correction phase within the NIT, expressed as the number of macroticks from the start of cycle. | 7 – 15 999 MT |
| *pRateCorrectionOut* | Magnitude of the maximum permissible rate correction value and the maximum drift offset between two nodes operating with non-synchronized clocks for one communication cycle. | 3 – 3 846 µT |
| *pSecondKeySlotID* | ID of the second key slot, in which a second startup frame shall be sent when operating as a coldstart node in a TT-L or TT-E cluster. If this parameter is set to zero the node does not have a second key slot. | 0 - *cStaticSlotIDMax* |
| *pTwoKeySlotMode* | Parameter indicating whether node operates as a coldstart node in a TT-E or TT-L cluster. If *pTwoKeySlotMode* is set to true then both *pKeySlotUsedForSync* and *pKeySlotUsedForStartup* shall also be set to true. If *pExternalSync* is set to true then *pTwoKeySlotMode* shall also be set to true. | Boolean |
| *pWakeupChannel* | Channel used by the node to send a wakeup pattern. *pWakeupChannel* shall be selected from among the channels configured by *pChannels*. | [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. | 0 – 63 [a] |

[a] A value of 0 or 1 prevents transmission of a wakeup pattern.

#### B.3.2.2 Protocol related

Protocol related node parameters are parameters that have a meaning in the context of the FlexRay protocol but are not used within the SDL models. They may have different values in different nodes of a cluster.

Table B.4 defines the local node protocol related parameters.

**Table B.4 — Local node protocol related parameters**

| Name | Description | Range |
|---|---|---|
| *pdMicrotick* | Duration of a microtick. | [0,0125, 0,025, 0,05] µs |
| *pNMVectorEarlyUpdate* | Parameter indicating when the update of the Network Management Vector in the CHI shall take place.<br>If *pNMVectorEarlyUpdate* is set to false, the update shall take place after the NIT.<br>If *pNMVectorEarlyUpdate* is set to true, the update shall take place after the end of the static segment. | Boolean |
| *pPayloadLengthDynMax* | Maximum payload length for dynamic frames.[a] | 0 - *cPayloadLengthMax* |
| *pSamplesPerMicrotick* | Number of samples per microtick. | [1, 2] |
| a For bit rates of less than 10 Mbit / s the payload length is restricted by the maximum transmission duration *adTxMax*. See B.4.42. | | |

### B.3.3 Physical layer parameters

For values of the following parameters please refer to ISO 17458-4.

Table B.5 defines the physical layer parameters.

**Table B.5 — Physical layer parameters**

| Name | Description |
|---|---|
| *dBDRx01* | Time by which a positive edge is delayed in a receiving node. |
| *dBDRx10* | Time by which a negative edge is delayed in a receiving node. |
| *dBDRxai* | Idle reaction time. Time by which a transmission becomes lengthened in a receiving node (when bus is switched from active to idle). If the last actively driven bit was HIGH the idle detection in the CC is not delayed. |
| *dBDTx01* | Time by which a positive edge is delayed in a transmitting node. |
| *dBDTx10* | Time by which a negative edge is delayed in a transmitting node. |
| *dBDTxRxai* | Delay between the rising edge of the TxEN signal at the BD when TxD is still low and the RxD signal at the BD going to high. |
| *dBDTxActiveMax* | Maximum duration of activation of a BD's TxEN input. |
| *dBDTxai* | Propagation delay of TxEN to bus activity on a transition from bus active to bus idle, i.e., the time by which a transmission becomes lengthened in a transmitting node when bus is switched from active to idle. |
| *dBDTxia* | Propagation delay of TxEN to bus activity on a transition from bus idle to bus active, i.e., the time by which a transmission becomes shortened in a transmitting node when bus is switched from idle to active. |

**Table B.5 —** *(continued)*

| Name | Description |
|---|---|
| *dBDTxDM* | Absolute time difference between *dBDTxia* and *dBDTxai*. |
| *dCCRxD01* | Time by which a rising edge on the CC's RxD pin is delayed by analogue effects inside the CC. |
| *dCCRxD10* | Time by which a falling edge on the CC's RxD pin is delayed by analogue effects inside the CC. |
| *dCCTxD01* | Delay for the rising edge of the TxD signal between the digital domain inside the CC and the output pin of the CC. |
| *dCCTxD10* | Delay for the falling edge of the TxD signal between the digital domain inside the CC and the output pin of the CC. |
| *dCCTxEN01* | Delay for the rising edge of the TxEN signal between the digital domain inside the CC and the output pin of the CC. |
| *dBranchRxActiveMax* | Maximum duration of activity of an incoming branch of an active star. |
| $dFrameTSSEMIInfluence_{M,N}$ | Assumed maximum shortening or lengthening of the TSS as a result of the influence of EMI effects on a TSS that is transferred from node M to node N (as a result, this term depends on the number of physical communication links, and thus the number of stars, between nodes M and N). A positive value indicates a length ening of the TSS. This value corresponds to the ISO 17458-4 parameters *dFrameTSSEMIInfluence0AS*, *dFrameTSSEMIInfluence1AS*, or *dFrameTSSEMIInfluence2AS*, depending on the number of active stars on the path between node M and node N.<br><br>For the purposes of the calculations of parameter ranges, this specification assumes the minimum and maximum values from a two star system, even though ISO 17458-4 places restrictions on such systems at certain bit rates. |
| $dFrameTSSLength Change_{M,N}$ | Amount by which the TSS is shortened or lengthened by the network (including active stars) for a frame sent from node M to node N. This parameter does not include the stochastic effects of EMI. A positive value indicates a lengthening of the TSS. This value corresponds to the ISO 17458-4 parameters *dFrameTSSLengthChange0AS*, *dFrameTSSLengthChange1AS*, or *dFrameTSSLengthChange2AS*, depending on the number of active stars on the path between node M and node N.<br><br>For the purposes of the calculations of parameter ranges, this specification assumes the minimum and maximum values from a two star system, even though ISO 17458-4 places restrictions on such systems at certain bit rates.[a] |
| $dPropagationDelay_{M,N}$ | Propagation delay from the TxD input pin of the transmitting BD (TP1_BD) of node M to the RxD output pin of the receiving BD (TP4_BD) of node N. |
| *dRing* | Duration of ringing on one segment of the network. If CAS / MTS or WUS symbols are transmitted, this value is used to calculate the time by which idle detection is delayed compared with a network without ringing. |
| $dRingRxD_{M,N}$ | Maximum time including the duration of ringing and idle reaction times when ring ing occurs after the transmission of a frame or WUDOP, between node M to node N. This value corresponds to the ISO 17458-4 parameters $dRingRxD_0$, $dRingRxD_{AS1}$, $dRingRxD_1$, $dRingRxD_{AS2}$ or $dRingRxD_2$, depending on the number of active stars on the path between node M and node N.<br><br>For the purposes of the calculations of parameter ranges, this specification assumes the maximum value from a two star system, even though ISO 17458-4 places restrictions on such systems at certain bit rates. |

a The frame decoding mechanism, a portion of which is described in Figure 64, relies on the assumption that the effects of the TSS length change and the TSS EMI influences, combined with the appropriate bit strobing and quantization effects, can increase the receiver's perspective of the length of a TSS by at most two bit times. The maximum values for *dFrameTSSLengthChange* and *dFrameTSSEMIInfluence* given in ISO 17458-4 satisfy this assumption.

**Table B.5 —** *(continued)*

| Name | Description |
|---|---|
| *dRxUncertainty* | Time following the end of a transmission where instability may occur on RxD as a result of echoes and or ringing. During this time the RxD output may change states several times and may not reflect the actual condition of the bus. |
| $dStarDelay01_k$ | Time by which a rising edge is delayed by star k. |
| $dStarDelay10_k$ | Time by which a falling edge is delayed by star k. |
| *dStarFES1LengthChange* | Length change of the last bit of a frame that passes through an active star. |
| $nStarPath_{M,N}$ | Number of stars on the signal path from any node M to a node N in a network with active stars. |
| *dStarSymbolEndLength Change* | Time by which the edge from low to idle after a symbol transmission is delayed by the effects of idle reaction times of the active star when a symbol passes through an active star. This time does not include the propagation delay inside the active star. |
| *dStarTSSLengthChange* | Frame TSS length change caused by an active star. |
| *dStarTxRxai* | Delay between the rising edge of the TxEN signal at the active star's CC interface when TxD is still low and the RxD signal of the active star's CC interface going to high. |
| $dSymbolEMIInfluence_{M,N}$ | Assumed maximum shortening or lengthening of the symbol as a result of the influence of EMI effects on a symbol that is transferred from node M to node N (as a result, this term depends on the number of physical communication links, and thus the number of stars, between nodes M and N). A positive value indicates a lengthening of the symbol. This value corresponds to the ISO 17458-4 parameters *dSymbolEMIInfluence0AS*, *dSymbolEMIInfluence1AS*, or *dSymbolEMIInfluence2AS*, depending on the number of active stars on the path between node M and node N.<br>For the purposes of the calculations of parameter ranges, this specification assumes the minimum and maximum values from a two star system, even ISO 17458-4 places restrictions on such systems at certain bit rates. |
| $dSymbolLengthChange_{M,N}$ | Amount by which a symbol is shortened or lengthened by the network (including active stars) for a symbol sent from node M to node N. This parameter does not include the stochastic effects of EMI. A positive value indicates a lengthening of the symbol. This value corresponds to the ISO 17458-4 parameters *dSymbolLengthChange0AS*, *dSymbolLengthChange1AS*, or *dSymbolLengthChange2AS*, depending on the number of active stars on the path between node M and node N.<br>For the purposes of the calculations of parameter ranges, this specification assumes the minimum and maximum values from a two star system, even though ISO 17458-4 places restrictions on such systems at certain bit rates. |
| $dWU_{0Detect}$ | Acceptance timeout for detection of a LOW phase in a wakeup pattern. The maximum value of this parameter represents a duration that will be accepted as a LOW phase by all BDs. |
| $dWU_{IdleDetect}$ | Acceptance timeout for detection of an Idle phase in a wakeup pattern. The maximum value of this parameter represents a duration that will be accepted as an Idle phase by all BDs. |
| $dWU_{Timeout}$ | Acceptance timeout for wakeup pattern recognition. The minimum value of this parameter represents a pattern duration that would be accepted as a wakeup by all BDs. |

### B.3.4 Auxiliary parameters

The following parameters are only introduced for configuration constraints.

Table B.6 defines the auxiliary parameters for configuration constraints.

**Table B.6 — Auxiliary parameters for configuration constraints**

| Name | Description |
|---|---|
| *aAssumedPrecision* | Assumed precision of the application network. |
| *aBestCasePrecision* | Upper bound for the clock deviation between two nodes (precision) assuming no faults are present in the cluster. |
| *adActionPointDifference* | Amount by which the static slot action point offset is greater than the minislot action point offset (zero if static slot action point is smaller than minislot action point). |
| *adBitMax* | Maximum bit time taking into account the allowable clock deviation of each node. |
| *adBitMin* | Minimum bit time taking into account the allowable clock deviation of each node. |
| *adDTSLow* | Duration of the low phase of the Dynamic Trailing Sequence. |
| *adInitializationErrorMax* | The maximum initialisation error that shall be tolerated by an integrating node. |
| *adInternalRxDelay* | Additional implementation dependent delay on the receive path of the decoder up to the strobe point, i. e. an additional synchronisation unit in front of the majority voting mechanism.<br>It is in the responsibility of the semiconductor manufacturer to specify this implementation dependent value for its devices. |
| $adLineDelay[Ch]_{M,N}$ | The contribution of the propagation delay attributed to the path lengths and the specific line delays $T'_0$ of the various segments of the communication path between node M and node N (see ISO 17458-4). This value also includes delays attributed to the circuit board traces between the CC and the BD in both the transmitter and the receiver. |
| *adMaxIdleDetectionDelayAfterHIGH* | Maximum time by which idle detection is delayed when ringing occurs after the transmission of a frame or WUDOP. |
| *adMicrotickDistError* | Maximum time difference in a node arising from the integral (i.e., non-fractional) distribution of microticks across different macroticks vs. an ideal distribution that allowed fractional microticks. The value is based on the local microtick *pdMicrotick*. |
| *adMicrotickMax* | Maximum microtick length of all microticks configured within a cluster. |
| *adMicrotickMaxDistError* | Maximum time difference in a cluster arising from the integral (i.e., non-fractional) distribution of microticks across different macroticks vs. an ideal distribution that allowed fractional microticks. The value is based on the largest microtick in the cluster, *adMicrotickMax*. |
| *adOffsetCorrection* | The duration in macroticks of the offset correction phase of the NIT. |
| *adPropagationDelayMax* | Maximum propagation delay of a cluster. |
| *adPropagationDelayMin* | Minimum propagation delay of a cluster. |
| *adRemOffsetCalculation* | Time after the beginning of the NIT necessary to ensure completion of the offset correction calculation. |
| *adRemRateCalculation* | Time after the beginning of the NIT necessary to ensure completion of the rate correction calculation. |

**Table B.6 —** *(continued)*

| Name | Description |
|---|---|
| *adSymbolWindowGuardInterval* | Required period of inactivity to ensure that symbols transmitted in the symbol window are perceived by all receivers as beginning in the symbol window. |
| *adTxDyn* | Upper bound on the duration of a dynamic frame transmission. |
| *adTxMax* | Maximum transmission duration of a CC (expressed in µs). |
| *adTxStat* | Upper bound on the duration of a static frame transmission. |
| *aFrameLength* | Frame length in bits of a frame including transmission start sequence, frame start sequence and frame end sequence but without idle detection time. |
| *aFrameLengthDynamic* | Frame length in bits of a dynamic frame (see *aFrameLength*). |
| *aFrameLengthStatic* | Frame length in bits of a static frame (see *aFrameLength*) |
| *aMicroPerMacroNom* | The nominal number of microticks per macrotick for a node. Note that this number need not be an integer. The allowable range for this auxiliary variable is bounded by *cMicroPerMacroNomMin* and *cMicroPerMacroNomMax*. |
| *aMinislotPerDynamicFrame* | Number of minislots needed to transmit a frame in the dynamic segment. |
| *aMixedTopologyError* | A term that expresses in microseconds the effect that the difference in propagation delay between the time source and time sink clusters has on the precision of the time sink cluster. |
| *aNegativeOffsetCorrectionMax* | The maximum amount of time by which the offset correction phase of the NIT might need to be shortened to account for negative offset corrections. |
| $anRingPath_{M,N}$ | Number of segments between node M and node N where ringing occurs. |
| *aOffsetCorrectionMax* | Cluster global magnitude of the maximum necessary offset correction value. |
| *aPayloadLength* | Payload length in two-byte words. |
| *aPayloadLengthDynamic* | Payload length in two-byte words of a dynamic frame. |
| *aPositiveOffsetCorrectionMax* | The maximum amount of time by which the offset correction phase of the NIT might need to be lengthened to account for positive offset corrections. |
| *aSinkPrecision* | An upper bound for the clock deviation between two nodes in the time sink cluster in a TT-E system. Different definitions of this parameter exist depending on the nature of the time source cluster. |
| *aWorstCasePrecision* | Upper bound for the clock deviation between two nodes (precision) when a limited number of Byzantine faults are present in the clock synchronisation of the cluster (see [11]). |



## B.4 Calculation of configuration parameters for nodes in a TT-D cluster

### B.4.1 General

This subclause describes how the configuration parameters are calculated for nodes operating in TT-D cluster. Unless specified differently in B.5, B.6, or B.7 these calculations also apply for nodes operating in TT-L or TT-E clusters.

### B.4.2 gClockDeviationMax

Clock sources for communication controllers in a cluster deviate from each other. The parameter *gClockDeviationMax* defines the maximum clock frequency deviation of any node in the cluster from the node's nominal clock frequency, defined as a ratio of the absolute value of the maximum frequency deviation to the nominal clock frequency, i.e., the actual clock frequency will be in the range as defined in the equation **(1)**.