MOST

Media **O**riented **S**ystems **T**ransport

Multimedia and Control Networking Technology

MOST Specification
Rev. 3.0 E2
07/2010

Legal Notice

COPYRIGHT

LICENSE DISCLAIMER

Nothing on any MOST Cooperation Web Site, or in any MOST Cooperation document, shall be construed as conferring any license under any of the MOST Cooperation or its members or any third party's intellectual property rights, whether by estoppel, implication, or otherwise.

CONTENT AND LIABILITY DISCLAIMER

MOST Cooperation or its members shall not be responsible for any errors or omissions contained at any MOST Cooperation Web Site, or in any MOST Cooperation document, and reserves the right to make changes without notice. Accordingly, all MOST Cooperation and third party information is provided "AS IS". In addition, MOST Cooperation or its members are not responsible for the content of any other Web Site linked to any MOST Cooperation Web Site. Links are provided as Internet navigation tools only.

MOST COOPERATION AND ITS MEMBERS DISCLAIM ALL WARRANTIES WITH REGARD TO THE INFORMATION (INCLUDING ANY SOFTWARE) PROVIDED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. Some jurisdictions do not allow the exclusion of implied warranties, so the above exclusion may not apply to you.

In no event shall MOST Cooperation or its members be liable for any damages whatsoever, and in particular MOST Cooperation or its members shall not be liable for special, indirect, consequential, or incidental damages, or damages for lost profits, loss of revenue, or loss of use, arising out of or related to any MOST Cooperation Web Site, any MOST Cooperation document, or the information contained in it, whether such damages arise in contract, negligence, tort, under statute, in equity, at law or otherwise.

FEEDBACK INFORMATION

Any information provided to MOST Cooperation in connection with any MOST Cooperation Web Site, or any MOST Cooperation document, shall be provided by the submitter and received by MOST Cooperation on a non-confidential basis. MOST Cooperation shall be free to use such information on an unrestricted basis.

TRADEMARKS

MOST Cooperation and its members prohibit the unauthorized use of any of their trademarks. MOST Cooperation specifically prohibits the use of the MOST Cooperation LOGO unless the use is approved by the Steering Committee of MOST Cooperation.

SUPPORT AND FURTHER INFORMATION

For more information on the MOST technology, please contact:

MOST Cooperation
Administration
Bannwaldallee 48
D-76185 Karlsruhe
Germany

Tel: (+49) (0) 721 966 50 00
Fax: (+49) (0) 721 966 50 01
E-mail: contact@mostcooperation.com
Web: www.mostcooperation.com

Contents

Document History

Changes MOST Specification Rev. 3.0 E1 to MOST Specification Rev. 3.0 E2

Change Ref.	Section	Changes
3V0E2_001	General	– Removed red change markers of revision 3.0 E1.
3V0E2_002	History	– Removed 3V0_047 because the change had been reverted. – Renumbered duplicate change refs. 3V0_132 and 3V0_133 to 3V0_164 and 3V0_165, respectively.
3V0E2_003	Glossary	– Distinguish between ConnectionMaster FBlock and Connection Manager. – Decentral Registry no longer “implemented in each node” because it does not necessarily exist in the NetworkMaster node. – Distinguish between System FBlocks and Application FBlocks.
3V0E2_004	1.2	– Removed distinction between oPhy and ePhy.
3V0E2_005	2.1.2.1	– Replaced “Function Block” with “FBlock” in Figure 2-1.
3V0E2_006	2.1.2.2	– Replaced “Function Block” with “FBlock” in Figure 2-2.
3V0E2_007	2.1.2.6	– Added functional address to list of MOST address types.
3V0E2_008	2.2.1	– Modified description of CD changer example so that “CDC” and “HMI” can no longer be mistaken as MOST node position addresses.
3V0E2_009	2.2.3.2	– The ConnectionMaster FBlock (0x03) is no longer mandatory. – Added FBlockID 0x09 DebugMessages and FBlockID 0x46 ESDR to Table 2-1.
3V0E2_010	2.2.3.4	– It can be determined for each function whether it is mandatory, optional, or conditional.
3V0E2_011	2.2.3.5.1	– Added group headings in Table 2-5 and Table 2-6.
3V0E2_012	2.2.3.5.2	– Swapped sections that describe methods with and without SenderHandle. Sender Handle methods are now described first.
3V0E2_013	2.2.3.5.4	– Consolidated description of $t_{ProcessingDefault1}$ and $t_{ProcessingDefault2}$. – Deviations from the default $t_{Processing}$ values do not have to be documented exclusively in the FBlock Specification anymore. – Limited the statement that the use of Start and StartResult OPTypes is not recommended to Application FBlocks.
3V0E2_014	2.2.3.8	– Modiefied data types overview to match the order of the corresponding sections.
3V0E2_015	2.2.3.8.11	– Added distinction between unstructured and structured streams, as well as simple and complex streams.
3V0E2_016	2.2.3.8.11.1	– Moved parts of the stream case introduction to the Stream section. – The selector of a complex stream may be embedded in another stream. – A string selector has to be coded with string code 0x01 (ISO8859/15 8bit).
3V0E2_017	2.2.4	– If MOST High Protocol syntax errors and application errors are reported, the Control Channel must be used.
3V0E2_018	2.2.4.2.3	– The complete DynamicArray is transferred to the notified Controllers when a Tag value is changed (one entry replaced by another).
3V0E2_019	2.2.5	– Notification for group addresses is explicitly limited to those that do not depend on dynamic address calculation. – It is no longer required that Notification.Set messages are limited to one telegram.
3V0E2_020	3.1.1.6	– The isochronous transmission class does not provide error detection (e.g., CRC).

Change Ref.	Section	Changes
3V0E2_021	3.1.1.7	– The maximum packet length for 16 bit addressing mode is now 1532 bytes. – Specifically described the extent of CRC protection on the Packet Data Channel.
3V0E2_022	3.1.2.1	– Added description of “listen only” nodes. – Removed superfluous sentences explaining “MOST nodes” and the nature of the following diagram.
3V0E2_023	3.1.2.2	– Added “OK” and “NotOK” substates in Figure 3-3 to link the System State to the NetInterface state.
3V0E2_024	3.1.2.2.2	– Corrected Init Error Shutdown condition for waking TimingSlaves. – Modified Figure 3-4 so that StableLockOccurrence and “closed ring” are included. – Modified Figure 3-6 so that StableLockOccurrence is included.
3V0E2_025	3.1.2.2.3	– Modified Figure 3-7 so that initially the node address is recalculated and ShutDown is used without SenderHandle.
3V0E2_026	3.1.2.2.5	– Clarified that “Diagnosis Result Ready” is caused by the completion of RBD Phase 3.
3V0E2_027	3.1.2.3.1	– Removed PermissionToWake property. – Removed CapabilityToWake.
3V0E2_028	3.1.2.3.2	– The ShutDown method does not use OPTypes with SenderHandle anymore. – The PowerMaster may override suspend requests from its Slaves. – If the shutdown reason ceases to exist, the PowerMaster cancels the shutdown. – Removed statement about “parked vehicles” preventing shutdown. – Figure 3-10 and Figure 3-11 modified: “Off Request” and “Net Off” were substituted with more descriptive terms.
3V0E2_029	3.1.2.3.3	– In Device Shutdown, the NetBlock is only active with limited functionality. – The ShutDown method does not use OPTypes with SenderHandle anymore. – Added note that in Device Shutdown the shutdown reason cannot be stored.
3V0E2_030	3.1.2.3.3.1	– The ShutDown method does not use OPTypes with SenderHandle anymore. – PowerMaster waits for $t_{WaitSuspend}$ rather than $t_{Suspend}$. – Improved description of potential repetitions before $t_{WaitSuspend}$ expires. – Devices with system functions shall reject requests to perform a Device Shutdown. – Made statement regarding streaming outputs more specific.
3V0E2_031	3.1.2.3.3.2	– The ShutDown method does not use OPTypes with SenderHandle anymore. – A device that is woken from Device Shutdown must perform the complete SystemCommunicationInit procedure.
3V0E2_032	3.1.2.3.3.3	– If the PowerMaster application is reset, it wakes all devices that are in Device Shutdown.
3V0E2_033	3.1.3.1.1	– Removed differentiation between the NetworkMaster having and not having a valid address at system startup because the node address is not stored persistently anymore.
3V0E2_034	3.1.3.2	– The ShutDown method does not use OPTypes with SenderHandle anymore. – Default transition made more generic by removing “Init Ready” label in Figure 3-12 – Caption for Figure 3-12 changed to refer to the NetInterface Normal Operation state.
3V0E2_035	3.1.3.2.1	– The ShutDown method does not use OPTypes with SenderHandle anymore.
3V0E2_036	3.1.3.2.2	– Consolidated description of actions that are taken as a result of Configuration.Status(NotOK).
3V0E2_037	3.1.3.3.1.2	– Consolidated description of actions that are taken as a result of Configuration.Status(NotOK).
3V0E2_038	3.1.3.3.2.4	– If the NetworkMaster receives CentralRegistry.Get in System State NotOK, it broadcasts Configuration.Status(NotOK).

Change Ref.	Section	Changes
3V0E2_039	3.1.3.3.5.4	– In the case of duplicate InstIDs, the FBlock that is already registered remains unchanged. – If the request to change the InstID is not successful, either the FBlock or the device is excluded from the Central Registry. – Added Figure 3-14 that specifies how the NetworkMaster resolves InstID conflicts.
3V0E2_040	3.1.3.3.6.1	– Added optional “own configuration invalid” handling.
3V0E2_041	3.1.3.3.6.2	– Removed redundant requirement that Configuration.Status with an empty list must be sent when the registry is full. This is already covered by section 3.1.3.3.6.3.
3V0E2_042	3.1.3.3.7.2	– The NetworkMaster has to reside in the same device as the TimingMaster. – Removed redundant requirement referencing $t_{MPRDelay}$.
3V0E2_043	3.1.3.3.7.3	– The NetworkMaster shall request ImplFBlockIDs exactly once from every device after System State OK was entered.
3V0E2_044	3.1.3.4.1	– The NetworkMaster device is not required to build a Decentral Registry.
3V0E2_045	3.1.3.4.2.3	– Completely revised how NetworkSlaves derive the NetworkMaster address.
3V0E2_046	3.1.3.4.3.8	– Added new figure “NetworkSlave determines the System State in NetInterface Normal Operation”.
3V0E2_047	3.1.3.4.3.13	– Consolidated description of actions that are taken as a result of Configuration.Status(NotOK).
3V0E2_048	3.1.3.4.3.15	– Removed remark regarding routing table and allocation table. – If a device receives a Configuration.Status(Invalid) with (part of) its own FBlockID(s) and InstID, it has to reinitialize all applications of the device. – Added optional “own configuration invalid” handling. – Stated more precisely what actions are required when reinitializing.
3V0E2_049	3.1.3.4.5	– Distinguished the roles of “NetworkMaster” and “Controller” when requesting FBlockIDs.
3V0E2_050	3.1.4.1.1.1	– Moved RBD diagram and table for TimingSlave here from “All devices” section. – Modified RBD diagram: modified transition from RBD_S_Slave to Diagnosis Ready so that it also checks StableLockOccurrence.
3V0E2_051	3.1.4.1.1.2	– Moved RBD diagram and table for TimingMaster here from “All devices” section. – Modified RBD diagram: added RBD_M_Start self transition and modified transition to Diagnosis Ready.
3V0E2_052	—	– Deleted “All devices” section.
3V0E2_053	3.1.4.1.2	– Moved introduction from “All devices” section. – Modified RBD results table.
3V0E2_054	3.1.4.2	– Figure 3-21 modified so that the ShutDown method does not use OPTypes with SenderHandle anymore; added extra check “Cause was Signal Off?”
3V0E2_055	3.1.4.4	– The ShutDown method does not use OPTypes with SenderHandle anymore. – Introduced “Retimed Bypass Mode” and modified description accordingly.
3V0E2_056	3.1.5.1.1	– The ShutDown method does not use OPTypes with SenderHandle anymore.
3V0E2_057	3.1.5.1.2	– Rephrased description of waking the network.
3V0E2_058	3.1.5.2	– Reaction on unlock for sinks is now described more specifically.
3V0E2_059	3.1.5.3	– Reaction on NCE for sinks is now described more specifically.
3V0E2_060	3.1.5.5	– Reaction on undervoltage for sources and sinks is now described more specifically. – Redundancies removed.
3V0E2_061	3.1.5.6.2.1	– The ShutDown method does not use OPTypes with SenderHandle anymore.

Change Ref.	Section	Changes
3V0E2_062	3.1.5.6.2.2	– The ShutDown method does not use OPTypes with SenderHandle anymore.
3V0E2_063	3.1.5.6.3.2	– The ShutDown method does not use OPTypes with SenderHandle anymore. – Removed use of PermissionToWake.
3V0E2_064	3.2.2	– Table 3-16 expanded and moved here from 3.2.2.1 because it is not limited to 16 bit addressing anymore.
3V0E2_065	3.2.2.1	– The logical node address is initialized to the default value on each transition to NetInterfaceNormal Operation. – Removed statements about storing the group address between power cycles. – Blocking and unblocking broadcast messages can either be single transfer or segmented messages. – Added description of free-up message.
3V0E2_066	—	– Removed “Asigning Priority Levels” section.
3V0E2_067	3.2.3	– Added paragraph explaining the potential consequences of retransmissions.
3V0E2_068	3.2.5.2.1	– Modified Figure 3-27 and Figure 3-28 so that it is indicated that the max. size of the payload is L_{AMSmax}. – For segments with TelID 3, the available payload must not be entirely unused. – Added examples of segmented transfer with available payload partially and entirely used. – Added recommendation to use entire available payload for TelID 1 and 2 in Size-Prefixed Segmented Messages. – Added caption to Table 3-18.
3V0E2_069	3.2.6.1.1.1	– The maximum MOST High Data Area size for 16 bit addressing is now 1524.
3V0E2_070	3.2.5.2.1	– Added recommendation to use the entire available payload within Size-Prefixed Segmented Messages. – Added caption for Table 3-18: Use of the TelID field.
3V0E2_071	3.2.7.2.1.1	– Removed superfluous remark from the introduction.
3V0E2_072	3.2.8.2.1	– Renamed section to “Connection Manager”. – Distinguished between the Connection Manager, which is mandatory, and the ConnectionMaster FBlock. – If the Connection Manager implements the ConnectionMaster, that FBlock has to be included in the Central Registry.
3V0E2_073	3.2.8.2.2	– Distinguished between Connection Manager and ConnectionMaster.
3V0E2_074	3.2.8.2.4	– Added section “Establishing DiscreteFrame Isochronous Streaming Connections”
3V0E2_075	3.2.8.2.5	– Added section “Removing DiscreteFrame Isochronous Streaming Connections”
3V0E2_076	3.2.8.2.6	– Completely revised.
3V0E2_077	3.2.10	– Added captions for tables. – The ShutDown method does not use OPTypes with SenderHandle anymore. – Removed minimum and typical value for $t_{WaitBeforeScan}$; made it a constraint and changed meaning to distinguish between first scan and subsequent scans. – Added constraint $t_{WaitBeforeRescan}$. – Removed $t_{Suspend}$. – Removed $t_{WaitAfterNCE}$ inequation because $t_{MPRdelay}$ was removed. – Added footnote for $t_{ProcessingDefault1}$ and $t_{ProcessingDefault2}$ regarding changing the timeout value. – Modified description of $t_{WaitForProcessing1}$ and $t_{WaitForProcessing2}$. – Modified values for $t_{WaitAfterOvertempShutDown}$. – Removed $t_{MPRdelay}$ timer.

Change Ref.	Section	Changes
3V0E2_078		– Removed distinction between oPhy and ePhy.
3V0E2_079	5.1.1	– Modified Figure A-5-1 and Figure A-5-3 to always derive the logical node address and not use a stored address. – Modified Figure A-5-2 so that NewExt is used instead of New. – Removed figure “Address Conflict Resolution” because it already exists in the main body of the specification.
3V0E2_080	7.2	– Changed maximum size of a control data message from 65 to 69. – The Data Link Layer overhead size is now 12.
3V0E2_081	7.3	– Modified bandwidth calculations to cover MOST50 and MOST150. – Added *Table A-7-4: NetBlock.Boundary influence in a MOST50 system*.
3V0E2_082	8	– New chapter “Addendum A – MOST50 Adaptation”.

To facilitate the identification of changes and corrections in this errata version, the paragraphs that were changed substantially are marked with red vertical lines on both sides.

Changes MOST Specification Rev. 3.0 to MOST Specification Rev. 3.0 E1

Change Ref.	Section	Changes
3V0E1_001	General	– After introduction, now using term “FBlocks” instead of “function blocks”. – Unified spelling of MOST terms and system states. – Explictly named CRCs, based on the context (e.g., Control Channel CRC). – Spelled out MSB as “most significant bit” or “most significant byte”, depending on the context; the same for LSB. – Correction of clerical errors in spelling, grammar, and word use. – Changed erroneous uses of “groupcast address” to “group address”. – Changed occurrences of “single telegram” to “single transfer”. – NetInterfacePowerOff state is now called NetInterface Off. – The blocking broadcast address is used for NetworkMaster.Configuration.Status and NetBlock.Shutdown.StartAck messages.
3V0E1_002	History	– Modified two entries: 3V0_105 (“unchanged” replaced “empty”) and 3V0_154 (removal of timer $t_{MsgResponse}$ missing).
3V0E1_003	2.2.3.2	– Added FBlockID 0x45 (TPEG Tuner) to list of FBlockIDs.
3V0E1_004	2.2.3.5	– Added footnote, stating that OPType Error may be used in cases where the SenderHandle is not known or not applicable.
3V0E1_005	2.2.3.5.1	– Made description of ErrorCodes 0x06 and 0x07 more precise. – Modified Figure 2-16 so that the ErrorAck OPType may be used, as well.
3V0E1_006	2.2.3.8	– Only “any allowed combination” of data types can be transported.
3V0E1_007	2.2.3.8.11.2	– Modified description of stream signal construct.
3V0E1_008	2.2.3.8.13	– “Length of the Short Stream” becomes “Length of the Content”.
3V0E1_009	2.2.4	– In Table 2-7, Modes 1 and 2 were revised. – MOST High Protocol connections must not be created for the OPTypes Error and ErrorAck, among others. – MOST High Protocol connection errors may optionally be reported on the Control Channel.
3V0E1_010	2.2.4.1.5	– Corrected description—BoolField is a function class, not a variable.
3V0E1_011	2.2.4.2.4.2	– Corrected spelling of “MoveArrayWindwow” to “MoveArrayWindow”. – Corrected MoveArrayWindow function use: StartResultAck OPType is not defined, using StartAck instead. – Corrected SearchArrayWindow function use: StartAck OPType is not defined, using StartResultAck instead.
3V0E1_012	3.1.1	– Modified Figure 3-2 so that the isochronous data transport mechanisms are now called “DiscreteFrame Isochronous”, “A/V Packetized Isochronous”, and “QoS IP”.
3V0E1_013	3.1.1.6	– The isochronous data use cases are now called “DiscreteFrame Isochronous (Streaming)”, “A/V Packetized Isochronous (Streaming)”, and “QoS IP (Streaming)”.
3V0E1_014	3.1.2.2	– Modified Figure 3-3: NetInterfacePowerOff state is now called NetInterface Off. Added new transitions and unified spelling.
3V0E1_015	3.1.2.2.1	– Added event “Diagnosis Result Start”

Change Ref.	Section	Changes
3V0E1_016	3.1.2.2.2	– Modified description of cause for Init Ready in TimingSlave device: Stable Lock was recognized and the System Lock Flag is set. – Added footnote, stating that the “Shutdown” transition is not contained in Figure 3-3. – Modified Figure 3-6 so that the “no” transition from “Lock stable and ring closed?” now ends in front of “Deactivate bypass at recognized Lock”. – Added event “Init Diagnosis Start”. – Added remark that Init Error Shutdown Ready does not use the Shutdown Flag and $t_{SSO\_Shutdown}$ does not apply.
3V0E1_017	3.1.2.2.3	– Modified Figure 3-7 so that an unexpected “Net Off” event also ends in Error Shutdown. An “Off Request” includes the reception of ShutDown.StartAck(…, Execute).
3V0E1_018	3.1.2.2.4	– Added remark that Diagnosis Error Shutdown does not use the Shutdown Flag and $t_{SSO\_Shutdown}$ does not apply.
3V0E1_019	3.1.2.2.5	– Added remark that Diagnosis Result Ready does not use the Shutdown Flag and $t_{SSO\_Shutdown}$ does not apply. – Added caption for Table 3-7.
3V0E1_020	3.1.2.3.2	– Added footnote “The parameter SenderHandle of NetBlock.ShutDown.ResultAck is ignored by the NetworkMaster.” – Corrected timer name: $t_{WaitSuspend}$ is used, not $t_{Suspend}$. – Clarified that the PowerMaster sends ShutDown.StartAck(…, Execute) after $t_{WaitSuspend}$. – In Normal Shutdown, before the PowerMaster switches the modulated signal off, $t_{ShutDownWait}$ and $t_{SSO\_Shutdown}$ have to expire.
3V0E1_021	3.1.3.3.5.4	– Clarified that the InstID is selected for the last FBlock that was “to be registered”. – If the InstID cannot be changed, the NetworkMaster may repeat the attempt with the same InstID.
3V0E1_022	3.1.3.3.7.3	– Corrected FktID of function ImplFBlockIDs from 0x011 to 0x012. – Modified note regarding the reporting of an empty FBlock list. – Requesting ImplFBlockIDs is only required for devices that reported a non empty list. – Extended definition of DeviceAvail “Incomplete”.
3V0E1_023	3.1.3.4.1.1	– Made “Building a Decentral Registry” a heading again; was formatted as body text by mistake.
3V0E1_024	3.1.4.3	– Renamed “unique controller” to “central component” to match the corresponding specification.
3V0E1_025	3.1.3.4.4	– In Table 3-12, added line with FBlockID 0x00, which will prompt the NetworkMaster to send ErrorCode 0x06. Made description of FBlockID 0x01…0xFE in combination with InstID 0x00 more precise. – The NetworkMaster responds with an error for FBlockID/InstID combinations that do not exist. – It is now explicitly stated that the NetworkMaster responds with ErrorCode 0x06 for FBlockID=0xFF and InstID < 0xFF in Table 3-12.
3V0E1_026	3.1.3.4.6	– Only FBlocks that are listed in the Central Registry have to support function FktIDs (0x000).
3V0E1_027	3.1.4.1.1.3	– New states: RBD_S_NetOff3 in Table 3-13 and RBD_M_NetOff3 in Table 3-14. – Modified Figure 3-19 and Figure 3-20 so that they contain the new states.
3V0E1_028	3.1.5.1.1	– Modified description and added reference to SSO and CU section.
3V0E1_029	3.1.4.2	– Shutdown Flag is set within tShutDown after an SSO or CU. – Improved and corrected Figure 3-21.
3V0E1_030	3.1.5.1.1	– Modified description to include Shutdown Flag.

Change Ref.	Section	Changes
3V0E1_031	3.1.5.5	– Modified Figure 3-23 so that muting and demuting does not exclusively refer to synchronous signals and changed the name of state NetInterface PowerOff to NetInterface Off.
3V0E1_032	3.1.5.6.2.1	– Correction: NetBlock.ShutDown.ResultAck, not Result, is used to request the temperature shutdown.
3V0E1_033	3.1.5.6.2.2	– Added footnote “The parameter SenderHandle of NetBlock.ShutDown.ResultAck is ignored by the NetworkMaster.”
3V0E1_034	3.1.5.6.3	– Corrected formatting error where body text had become part of the heading.
3V0E1_035	3.2.2.1	– Added line for address 0x0000 in Table 3-17.
3V0E1_036	3.2.5.2.1	– Removed superfluous remark regarding the obligation to implement AMS. – Removed “Length” after “OPType” of “DeviceID.FBlockID.InstID.FktID.OPType (Parameter)” because the length is calculated and not explicitly transmitted. – Added restrictions regarding the use of TelID 4. – Added unnumbered heading “TelID, TelLen, and Data”. – TelIDs greater than 4 have to be ignored.
3V0E1_037	3.2.6.1	– Modified Figure 3-32 so that the isochronous data transport mechanisms are now called “DiscreteFrame Isochronous”, “A/V Packetized Isochronous”, and “QoS IP”.
3V0E1_038	3.2.7.1	– Modified Figure 3-33 so that the isochronous data transport mechanisms are now called “DiscreteFrame Isochronous”, “A/V Packetized Isochronous”, and “QoS IP”.
3V0E1_039	3.2.7.2.1	– Removed requirement that sources and sinks are numbered in ascending order without gaps.
3V0E1_040	3.2.9	– For timing constraints, the character “-” means “not defined”. – Modified description of $t_{ShutDown}$; the constraint also applies when setting the Shutdown Flag. – Made “Typ Value” of tWaitSuspend “System Integrator specific”. – Modified description of $t_{ShutDownWait}$ so that it refers to the Shutdown Flag. – Made “Typ Value” and “Max Value” of tSlaveShutdown “System Integrator specific”. – Removed inequation for $t_{MPRdelay}$. – Changed “Max Value” of $t_{WaitForProperty}$ to 400. – Made “Min Value” of tProcessingDefault1 “System Integrator specific”. – Made “Min Value” and “Typ Value” of tCM_DeadlockPrev “System Integrator specific”. – Modified most values for Ring Break Diagnosis. – Corrected description of $t_{Diag\_Slave\_T1}$.

To facilitate the identification of changes and corrections in this errata version, the paragraphs that were changed substantially are marked with red vertical lines on both sides.

Changes MOST Specification 2V5-00 to MOST Specification 3V0-00

Change Ref.	Section	Changes
3V0_001	General	– Specification structure modified to better distinguish between application layer and protocol layer. – Improved and unified the use of commonly used terms. – Enforced clear distinction between the terms “protocol”, “message”, and “function”. – Replaced occurrences of “physical address” with “node position address”. – Corrected clerical errors.
3V0_002	Glossary	– Removed references to specific speed grades. – Added entries for Notification, Notification Matrix, and OPType.
3V0_003	1.2	– Removed “MOST Hardware” from scope of the specification. – Added remark that the specification is speed grade independent and not fully backward compatible to existing speed grades MOST25/50 described in MOST Spec. Rev. 2.5. – Moved distinction between ePhy and oPhy here.
3V0_004	1.3	– Updated MOST Framework diagram.
3V0_005	1.4	– Added Unicode, IEEE 802.3, ISO 8859, IEC 62106, ETSI EN 300401 and Shift JIS to list of references.
3V0_006	1.5	– Removed distinction between MOST25 and MOST50. The specification is speed grade independent. – Removed note that the MOST Specification does not depend on any other MOST Cooperation documents.
3V0_007	2.1.1	– New section outlining a few basic principles of MOST.
3V0_008	2.1.1.1	– Renamed to “Overview of Data Transport”
3V0_009	2.1.1.1.1	– Removed note on recommended data rate limit for the Control Channel. – Removed partial description of Packet Data Channel because of redundancy.
3V0_010	2.1.1.1.2	– Made application scenario for streaming data more specific by mentioning time-synchronized transmission.
3V0_011	2.1.1.1.3	– Added remark that the Packet Data Channel is secured by CRC and has ACK/NAK with automatic retry.
3V0_012	2.1.2	– Stated more clearly that the section describes the logical model of a MOST device.
3V0_013	2.1.2.1	– Moved function interface description to appendix because function interfaces are now optional.
3V0_014	-	– Merged section “Managing Streaming/Packet Bandwidth” with “Bandwidth Management” section.
3V0_015	2.1.2.1.1	– Added remark that a Controller can be associated with more than one Slave and vice versa.
3V0_016	2.1.2.1.2	– Removed remark on using FBlock Shadows on the Controller side—implementation detail. – Removed distinction between properties and methods because these are introduced later.
3V0_017	2.1.2.2	– For clearer distinction, methods and properties are called “categories”, not “classes”.
3V0_018	2.1.2.3	– Removed overly extensive description of RDS example scenario. – Changed non-Ack OPTypes to OPTypes using Ack.
3V0_019	2.1.2.4	– Rephrased slightly misleading statement that properties represent limits.
3V0_020	2.1.2.5	– Added recommendation to change properties with Set instead of SetGet when using notification to avoid two identical status messages. – Removed definition example.

Change Ref.	Section	Changes
3V0_021	-	– Moved “Function Interfaces” section to appendix because function interfaces are now optional.
3V0_022	2.1.2.6	– Inserted consolidated list of address types, which includes Ethernet MAC address.
3V0_023	2.2	– Section renamed from “Protocol” to “Application Layer Protocol Specification”.
3V0_024	2.2.1	– Changed heading to “Application Messages in a MOST Network” – Added Packet Data Channel as potential channel for transport of application messages. Previously, only the Control Channel was mentioned.
3V0_025	2.2.2	– Added brief introductory description of properties and methods.
3V0_026	2.2.2.1	– Now using actual device names in the example instead of the generic terms Controller and Slave. – Added brief description of FBlock Shadow concept. – Consolidated description to reduce emphasis on implementation details. – Added note that a Status reply has the same functional address as the command, but a different DeviceID. – Removed overly implementation centric example. – In addition to FBlockID and InstID, the FktID is required to make an address unique.
3V0_027	2.2.2.2	– Added introduction to the use of methods.
3V0_028	2.2.2.2.1	– Renamed from “Special Case Using Routing” to “Using Methods with SenderHandle”. – Methods with SenderHandle are now the standard case. – SenderHandle concept is now described before being used in the example. – Added restriction that SenderHandle must not contain application relevant information. – Replaced “Controller” and “Slave” with actual devices names used in the example.
3V0_029	2.2.2.2.2	– Renamed from “Standard Case” to “Using Methods Without SenderHandle”. – Completely revised. – Added note that methods without SenderHandle are deprecated.
3V0_030	-	– Removed redundant section “Application Protocol Basics”.
3V0_031	2.2.3	– Renamed to “Structure of MOST Messages” – Removed “Length” from the structure because it is not transmitted as part of the message.
3V0_032	2.2.3.1	– Stated that the use of device address 0xFFFF is not available for communication directed from Slave to Controller.
3V0_033	2.2.3.1.1	– Moved section here. Previously under the “DeviceID” section.
3V0_034	2.2.3.2	– “Errors … for FBlockID 0xFF should not be returned” becomes “must not be returned”. – Removed name for FBlockID 0x00. Extended explanation. – FBlockID 0xC8 becomes “Reserved for Compliance Testing”. – Removed Secondary Node (FBlockID 0xFC), which is no longer covered by the specification. – Modified ranges from FBlockID 0xF0 to 0xFE. – Added Tool FBlock (0x0E), AuxiliaryOutput FBlock (0x25), and AuxiliaryInputOutput FBlock (0x29).
3V0_035	2.2.3.2.1	– Added section “Handling of Supplier Specific FBlocks” to improve document structure.
3V0_036	2.2.3.3.1	– Changed heading from “Responsibility” to “Uniqueness of Functional Addresses”

Change Ref.	Section	Changes
3V0_037	2.2.3.4	– Merged “Mandatory” and “Extensions” ranges into the “Application” range. – Added FBlockIDs and FunctionIDs ranges to table. – The requirement “a controller must verify the identity of a device before using any proprietary function of FBlock” is now a recommendation. – Added FBlockInfo as example for obtaining the identity of a device.
3V0_038	2.2.3.5	– Reintroduced ErrorAck OPType for properties (in conjunction with ErrorCode 0x01…0x04). – Added footnote that non-Ack OPTypes are deprecated. – Added footnote that GetInterface and Interface are optional OPTypes. – Added footnote that ErrorAck is required for properties to report syntax errors in conjuction with illegal OPTypes.
3V0_039	2.2.3.5.1	– Added reference to AMS section for details on Data[0]…Data[n]. – More precise specification of ErrorCode 0x0C, ErrorInfo 0x01 and 0x07. – Modified table entry for ErrorCode 0x20. – Rephrased note so that it cannot be misread as entirely dismissing error handling. – Slaves shall rely on ErrorCodes defined in MOST Specification. Added footnote that error messages are not limited to single telegrams (except segmentation errors). – No error reply for multicast messages (broadcast, groupcast, “all FBlocks”, “all instances”). – Removed overly detailed description of the segmented message concept. – Segmentation errors are no longer only directed from Slave to Controller – Removed MOST25 specific Secondary Node error. ErrorCode 0x0A is now reserved. – Removed Secondary Node branch from Figure 2-16. – Removed “Start Execution” action from Figure 2-16 because of potential conflicts in some scenarios. – Removed redundant example for ErrorCode 0x07.
3V0_040	2.2.3.5.2	– Added note stating that non-Ack OPTypes are no longer recommended. – Corrected description that correlated StartAck with ResultAck.
3V0_041	2.2.3.5.4	– Added remark that usage of Start and StartResult is no longer recommended. – Changed diagrams to use Ack OPTypes instead of non-Ack OPTypes and moved them to the corresponding section.
3V0_042	2.2.3.5.6	– Changed reference to Start OPType to StartAck because non-Ack OPTypes are no longer recommended.
3V0_043	2.2.3.5.7	– Removed ambiguous remark on independence from Notification Matrix.
3V0_044	2.2.3.5.8	– OPTypes GetInterface and Interface are now optional. Added corresponding note.
3V0_045	2.2.3.5.9	– Added note that usage of Increment and Decrement is not recommended for multicast messages.
3V0_046	2.2.3.5.10	– Added remark that usage of Abort is no longer recommended.
3V0_048	2.2.3.6	– Completely revised.
3V0_049	2.2.3.7	– Added remark that the system-specific maximum length is defined by the System Integrator. – Replaced remark about length not being transmitted directly with reference to AMS section.
3V0_050	2.2.3.8	– Removed limitation that messages are sent on the Control Channel only. – Removed data field length description. Already contained in previous section. – Removed recommendation to limit messages to single telegrams. – Added note, stating that the exponent of a value is not transmitted over MOST with the value.

Change Ref.	Section	Changes
3V0_051	2.2.3.8.2	– Inserted BitField/Array example that was previously contained in the BitSet function class description. – Corrected minor error in example (wrong bit value).
3V0_052	2.2.3.8.10	– “Unicode, UTF8” was labeled as not being ASCII compatible; corrected.
3V0_053	2.2.3.8.11	– Added list of restrictions for the use of data type Stream.
3V0_054	2.2.3.8.11.1	– New section: Stream Cases.
3V0_055	2.2.3.8.11.2	– New section: Stream Signals.
3V0_056	2.2.3.8.12	– Added statement that the Length field includes the MediaType. – Included examples.
3V0_057	2.2.3.8.13	– Added statement that short streams can be structured like streams.
3V0_058	2.2.4	– Emphasized that OPTypes GetInterface and Interface are optional. – Removed note that Unicode is not ASCII compatible. – Changed Channel Type table entries for ProcessingAck and ResultAck. – Removed reference to MOST High over Control Channel, which is no longer supported.
3V0_059	2.2.4.1	– Updated overview: Container function class now supports Stream, Classified Stream and Short Stream.
3V0_060	2.2.4.1.1	– Parameter “Boolean” becomes “Content”; the parameter name is no longer a type name. – Corrected parameter type for function class Switch to be Boolean. – Added footnote that Interface and GetInterface are optional OPTypes.
3V0_061	2.2.4.1.2	– Parameter “Number” becomes “Content”; the parameter name is no longer a type name. – Added list of allowed data types and description. – Added GByte, TByte, bit, bps, kbps, Mbps, Bps, kBps, and MBps to table of units. – Added remark that units are based on the International System of Units (SI).
3V0_062	2.2.4.1.3	– Parameter “String” becomes “Content”; the parameter name is no longer a type name. – Added allowed data type and description.
3V0_063	2.2.4.1.4	– Parameter “Pos” becomes “Content” to avoid confusion with Arrays. – Corrected parameter type to be Enum. – Removed Increment and Decrement OPTypes.
3V0_064	2.2.4.1.6	– Parameter “SetOfBits” becomes “Content” for reasons of consistency. – Removed example of BitSet with Array, which is not supported by this function class.
3V0_065	2.2.4.1.7	– Parameter “Classified Stream” becomes “Content” for reasons of consistency. – Container function class now supports Stream, Classified Stream and Short Stream. – Added DataType parameter to Interface OPType.
3V0_066	2.2.4.2	– Changed description of Sequence Property to indicate that it contains single value which can be treated as one property. – Added function class Map to function class table. – Added Unclassified Property to function class table. – Added remark that all types except “Stream” can be used as parameters in Arrays, Records and Sequences. Stream may be used as last element in Records or Sequences. – Added remark that OPTypes are omitted in the interface description of function classes with multiple parameters.
3V0_067	2.2.4.2.1	– Removed note about omission of OPTypes in function interfaces. – Added statement that an Array must be the last parameter if used within a Record. – Modified Figure 2-20 so that the Array is the last element.

Change Ref.	Section	Changes
3V0_068	2.2.4.2.2	– Removed references to Function Interfaces, which have become optional. – Specified that only Array elements that were changed are transmitted. – Changed InstID in examples from 0 to 0x01.
3V0_069	2.2.4.2.4	– Removed references to Function Interfaces, which have become optional. – Specified that the entire DynamicArray is transmitted when the size changes.
3V0_070	2.2.4.2.4.3	– Replaced wrong parameter FktIDMotherArray with the correct FktIDArrayWindow in function MoveArrayWindow.
3V0_071	2.2.4.2.4.4	– Added remark that resynchronization for multiple LongArray Controllers in one device is coordinated within the device.
3V0_072	2.2.4.2.5	– Changed non-Ack OPTypes to Ack OPTypes in the description of “Editing in Function Class Map” because the use of non-Ack OPTypes is no longer recommended.
3V0_073	2.2.4.2.6	– Added list of characteristcs, which distinguish Sequence Properties from Unclassified Properties. – Added remark that the parameters of the sequence do not have to be of the same type. – Added list of allowed data types and description. – Added footnote that a Stream parameter—if it exists—must be the last parameter.
3V0_074	2.2.4.3	– Modified description of Unclassified Method. – Changed non-Ack OPTypes to Ack OPTypes in the introduction because the use of non-Ack OPTypes is no longer recommended.
3V0_075	2.2.4.3.1	– Added description of SenderHandle parameter. – Added StartAck OPType.
3V0_076	2.2.4.3.2	– Added note that identical parameter lists for Set, SetGet and Status are not required for Sequence Methods. – Added remark that the parameters of the sequence do not have to be of the same type. – Added requirement that at least one parameter besides SenderHandle has to be present. – Added StartAck OPType – Added list of allowed data types and description of Parameter and SenderHandle.
3V0_077	2.2.5	– Configuration.Status(NewExt) replaces Configuration.Status(New). – Added description of the “implicit notification” concept. – Added statement indicating that Notification Matrix entries for invalid target addresses are removed in any case. This includes those that were set by the Notification function as well as those set through implicit notification. – Made restriction on maximum number of FktIDs per message more generic; the list has to fit into a single telegram. – More precise description of notification timings. – Removed note on notification of Arrays, DynamicArrays, and ArrayWindows. The slightly modified mechanism is now described in the respective sections.
3V0_078	-	– Removed Secondary Node and changed description accordingly.
3V0_079	3	– New introduction to Network Section. – Modified Figure 3-1: Application and Network Service of a MOST device.
3V0_080	3.1.1	– New section “MOST Data”.
3V0_081	3.1.1.1	– Compacted description of control data.
3V0_082	3.1.1.2	– Added explanation of the term “source data”.
3V0_083	3.1.1.5	– Renamed “Streaming Data” to “Synchronous Data” to emphasize contrast to isochronous data in this context. – Removed misleading remark about routing engine.

Change Ref.	Section	Changes
3V0_084	3.1.1.6	– New section “Isochronous Data”
3V0_085	3.1.1.7	– Completely reworked Packet Data section. – MAMAC is no longer included. – Low-level retries are supported for 16 and 48 bit addressing. The System Integrator determines the number of retries and the time between retries.
3V0_086	-	– Removed “Source Data Interface” section.
3V0_087	-	– Removed “Transparent Channels” section.
3V0_088	3.1.2.1	– Fixed clerical error in device states list: DeviceStandBy was missing. – Renamed Primary Node to MOST Node because the concept of Secondary Nodes has been abandoned. – Removed layer model diagram.
3V0_089	3.1.2.2	– Removed NetOn event and Application Init state from diagram. – Added optional state NetInterface Diagnosis Result in diagram.
3V0_090	3.1.2.2.2	– Removed MOST25 specific description of stable lock handling.
3V0_091	3.1.2.2.3	– Renamed “Net On” event to “Init Ready”. – Added remark that NetInterfaceNormalOperation is referred to as NetOn state. – Clarification: unlocks that are not critical unlocks do not result in an Error Shutdown.
3V0_092	3.1.2.2.4	– Added transition to state NetInterface Diagnosis Result. – Removed detailed description of Ring break diagnosis which is contained in a later section.
3V0_093	3.1.2.2.5	– Added section “NetInterface Diagnosis Result” to describe the corresponding state.
3V0_094	3.1.2.3.1	– Renamed “NetOn” event to “InitReady” in diagram.
3V0_095	3.1.2.3.2	– Replaced “electrical wak-up” with “any other wake-up trigger”. – Added distinction between Error Shutdown and Normal Shutdown to match use in other parts of the specification. – PowerMaster is now included: all devices switch the modulated signal off on certain errors. – Clarification: Error Shutdown is the result of a critical unlock, not a “normal” unlock. – Improved and more precise description. – ShutDown is now using Ack OPTypes because non-Ack OPTypes are no longer recommended
3V0_096	3.1.2.3.3	– NetBlock.Shutdown.Start becomes NetBlock.ShutDown.StartAck because non-Ack OPTypes are no longer recommended.
3V0_097	3.1.2.3.3.1 3.1.2.3.3.2	– ShutDown is now using Ack OPTypes because non-Ack OPTypes are no longer recommended.
3V0_098	3.1.3.1.1	– Removed statement that collisions will not occur more than once after change to NotOK. – Rephrased bulleted list so it becomes clear that the relevant address is that of the NetworkMaster.
3V0_099	3.1.3.2	– Removed transition Configuration.Status(OK) within System State OK. – Renamed NetOn event to Init Ready. – Figure 3-12—Shutdown.Start(Execute) becomes ShutDown.StartAck(…, Execute) because non-Ack OPTypes are no longer recommended.
3V0_100	3.1.3.2.1	– NetBlock.Shutdown.Start(Execute) becomes NetBlock.ShutDown.StartAck(…, Execute) because non-Ack OPTypes are no longer recommended.
3V0_101	3.1.3.2.2	– Removed OK transition from Table 3-9. – Removed requirement to route zeros. The sink does this now.

Change Ref.	Section	Changes
3V0_102	3.1.3.3.1	– Added recommendation to perform additional retries after failed Configuration.Status broadcast.
3V0_103	3.1.3.3.1.2	– Removed remark that Configuration.Status(NotOK) does not necessarily imply a state change. – Added remark that Configuration.Status(NotOK) has to be sent if the Central Registry becomes unavailable due to an internal NetworkMaster failure.
3V0_104	3.1.3.3.4.5	– Completely reworked.
3V0_105	3.1.3.3.5.4	– The NetworkSlave now responds with an unchanged list if the InstID cannot be changed. – The maximum number of NetworkMaster retries is defined by the System Integrator.
3V0_106	3.1.3.3.5.5	– Removed reference to Secondary Node.
3V0_107	3.1.3.3.6.1	– Modified description of DeltaFBlockIDList length requirement (must fit into a single telegram).
3V0_108	3.1.3.3.6.2	– Added description of NetworkMaster reaction when Central Registry is “full”. – Configuration.Status(NewExt) replaces Configuration.Status(New). – Modified description to also include DeviceID in FBlock list.
3V0_109	3.1.3.3.6.3	– Configuration.Status(NewExt) replaces Configuration.Status(New).
3V0_110	3.1.3.3.7.3	– New section “System Configuration Status Information”.
3V0_111	-	– Removed Secondary Node section.
3V0_112	-	– Removed Large Updates to the Central Registry section.
3V0_113	0	– Added remark that a NetworkSlave is responsible for successful transmission of messages that change the Central Registry.
3V0_114	3.1.3.4.4	– Removed potentially misleading introduction. – When an FBlock does not exist, the NetworkMaster now sends an empty Status message instead of an error. – Added description of NetworkMaster reaction to FBlockID/InstID combinations.
3V0_115	3.1.3.4.3.11	– Large Updates to the Central Registry no longer apply. Configuration.Status(OK) in System State OK is now ignored.
3V0_116	3.1.3.4.3.13	– Removed requirement to route zeros. The sink does this now.
3V0_117	3.1.3.4.3.14	– Heading changed to “Reaction to Configuration.Status(NewExt)”
3V0_118	3.1.3.4.7	– New section “Extended FBlock Identification”.
3V0_119	3.1.3.4.4	– Added footnote on NetworkMaster answer behavior if the NetworkSlave is not contained in the Central Registry. – Added remark on NetworkMaster behavior in case of omitted parameters. – New table “FBlockID, InstID combinations for querying the Central Registry”.
3V0_120	3.1.4.1	– New section “Ring Break Diagnosis”.
3V0_121	3.1.4.2	– New section “Detection of Sudden Signal Off and Critical Unlock”.
3V0_122	3.1.4.3	– New section “Shutdown Result Analysis”.
3V0_123	3.1.4.4	– New section “Coding Error Counter”.
3V0_124	3.1.5	– Consolidated description of error handling.
3V0_125	3.1.5.1	– Removed misleading examples of devices (transceiver…); those were only parts of devices.

Change Ref.	Section	Changes
3V0_126	3.1.5.1.1	– Consolidated description of modulated signal off handling. – Shutdown.Start (Execute) becomes Shutdown.StartAck(…, Execute) because non-Ack OPTypes are no longer recommended.
3V0_127	3.1.5.4.1	– Added note that the device must not communicate until it receives Configuration.Status after an internal failure.
3V0_128	3.1.5.4.2	– Added remark that a device must not attempt to re-register before deregistration was successfully transmitted. – Added remark that the NetworkMaster has to process FBlockIDs.Status messages in the order they are received. – Replaced redundant description of DeltaFBlockList length limitations with references. – Control Enum modified: Value 0x03 (New) becomes “Reserved”; 0x04 (NewExt) added.
3V0_129	3.1.5.5	– Renamed “Supply Voltage” section to “Undervoltage Management” – Changed transition from DevicePowerOff to DeviceStandBy (U > U*) in Figure 3-21. – Added footnote, explaining U*.
3V0_130	3.1.5.6	– Over-Temperature management section completely restructured. It now clearly distinguishes between mandatory and option behavior, as well as Slave and PowerMaster behavior. – It is now explicitly stated that the use of the PermissionToWake property is optional.
3V0_131	3.1.5.6.2.1 3.1.5.6.3.2	– NetBlock.ShutDown is now using Ack OPTypes because non-Ack OPTypes are no longer recommended.
3V0_132	3.2	– New section “Network Layer Protocol Specification”.
3V0_133	3.2.2	– Combined addressing topics in this section and added Ethernet addressing as well as unblocking broadcast address.
3V0_164	3.2.3	– Section on priority levels completely reworked.
3V0_165	3.2.4	– Removed reference to storing Error_NAK. – Heading changed to “Handling Overload in a Message Receiver” – “Sinks” become “message receivers” to prevent confusion with streaming data.
3V0_134	3.2.5.1	– Updated diagram and description. Maximum number of data bytes for CMS now given as L_{CMSmax}. the maximum length of 51 bytes for MOST150 was moved to a footnote to provide an example.
3V0_135	3.2.5.2	– Completely reworked and added Size-Prefixed Segmented Transfer for TelID 4. – Removed TelIDs 8 and 9; MOST High over Control Channel is no longer supported.
3V0_136	3.2.6	– Removed remark that ACK/NAK mechanisms are not required for packet data. – Added remark that the Packet Data Channel can also be used for control data.
3V0_137	3.2.6.1	– Figure 1-1: Removed MOST High over Control Channel and added isochronous data.
3V0_138	3.2.6.1.1	– Renamed section from “Securing Data” to “Use Cases and Data Formats”. – Removed diagram for “data link layer 48 bytes mode”. – Removed MAMAC parts. – Added Ethernet over MOST.
3V0_139	3.2.6.1.1.1	– Added MOST High description that was previously in the AMS section. – Added missing Source Address to MOST High diagram.
3V0_140	3.2.7.1	– Removed reference to speed grade MOST25. – Figure 3-33: Removed MOST High over Control Channel and added isochronous data. – Removed notion of streaming channels building a connection.

Change Ref.	Section	Changes
3V0_141	3.2.7.2.1	– Renamed SyncDataInfo to StreamDataInfo. – SourceCount, SinkCount substituted with SourceNrList, SinkNrList.
3V0_142	-	– Deleted section describing the SourceHandles function; SourceHandles is no longer supported.
3V0_142	3.2.7.2.1.1	– Removed SourceConnect/SourceDisconnect approach, which is specific to MOST25. – SourceActivity, Allocate, and DeAllocate are now using Ack-OPTypes. – Changed parameter list for SourceInfo: BlockWidth and ConnectionLabel are now parameters. – Changed parameter list for Allocate: BlockWidth and ConnectionLabel replace ChannelList. Parameter SrcDelay was removed.
3V0_143	3.2.7.2.1.2	– Connect and Disconnect are now using Ack-OPTypes. – Changed parameter list for SinkInfo: BlockWidth and ConnectionLabel are now parameters. – Changed parameter list for Connect: BlockWidth and ConnectionLabel replace Channels. Parameter SrcDelay was removed.
3V0_144	3.2.7.2.1.3	– Removed SourceConnect/SourceDisconnect approach, which is specific to MOST25. – Removed reference to SrcDelay. – Rephrased so that connections and connection labels are used instead of channels.
3V0_145	-	– Section “Order of Streaming Channel Lists” was removed because channel lists are no longer relevant.
3V0_146	3.2.7.2.2	– Removed MOST25 specific passage on delay compensation.
3V0_147	3.2.8.1	– Merged bandwidth management description from other parts of the document into this section. – MoveBoundary is now using StartResultAck instead of StartResult. – Connections are not removed anymore when changing the Boundary.
3V0_148	3.2.8.2.1	– Renamed BuildSyncConnection to BuildConnection. – Renamed SyncConnectionTable to ConnectionTable. Replaced ChannelList parameter with BlockWidth and ConnectionLabel. Removed NoChannels parameter. – Renamed RemoveSyncConnection to RemoveConnection.
3V0_149	3.2.8.2.2	– Removed SourceConnect/SourceDisconnect approach, which is specific to MOST25. – Renamed BuildSyncConnection to BuildConnection. – Removed SrcDelay parameter. – Replaced ChannelList with BlockWidth and ConnectionLabel. – All methods are now using Ack OPTypes because non-Ack OPTypes are no longer recommended.
3V0_150	3.2.8.2.3	– Removed SourceConnect/SourceDisconnect approach, which is specific to MOST25. – Renamed RemoveSyncConnection to RemoveConnection. – All methods are now using Ack OPTypes because non-Ack OPTypes are no longer recommended.
3V0_151	3.2.8.2.6	– Added remark that the validity of a connection is verified by the data link layer. A source malfunction leads to automatic de-allocation. At the same time, affected applications detect the disappearance of the streaming signal and secure their output.
3V0_152	-	– Removed section “Enabling Streaming Output”; it is no longer relevant.
3V0_153	-	– Removed section “Source Drops”; it is no longer relevant.

Change Ref.	Section	Changes
3V0_154	3.2.9	– Removed timer $t_{MsgResponse}$. – Removed MOST25 specific timer $t_{Boundary}$. – Modified t_{Config} values. – Corrected description of t_{WakeUp}, which was using $t_{Boundary}$. – Removed $t_{WaitBeforeScan}$ maximum value. – Raised maximum value of $t_{DelayCfgRequest1}$ from 550 to 700. – Modified $t_{ShutDown}$ values. – Modified $t_{Suspend}$ values and description. – Added $t_{WaitSuspend}$. – Modified $t_{RetryShutDown}$ values. – Modified $t_{Restart}$ values, changed description. – Modified t_{Lock} values, changed description. – Removed timer $t_{ResourceRetry}$. – Maximum value and description of $t_{WaitAfterNCE}$ changed. – Improved description of t_{Answer}. – Added footnote for $t_{WaitForProperty}$ (max. value depends on retry settings). – Removed timer $t_{CleanChannels}$ – Added timer $t_{SSO\_Shutdown}$ – Modified all ring break diagnosis timers. – ShutDown is now using Ack OPTypes because non-Ack OPTypes are no longer recommended.
3V0_155	-	– Merged section “Master/Slave” into introduction.
3V0_156	-	– Removed “Automatic Allocation Mechanism” section.
3V0_157	-	– Removed entire hardware chapter.
3V0_158	4	– New appendix “Optional OPTypes”.
3V0_159	5	– Network Initialization chapter is now tagged “informative”.
3V0_160	5.1.1	– Changed all occurrences of “NetOn” to “Init Ready”. – Figure A-5-1 now using Configuration.Status(NewExt).
3V0_161	6	– Removed MOST25 specific parts. – Changed data rate example to contain MOST150 values.
3V0_162	7	– Sample frequency no longer in a range between 30 kHz and 50 kHz but either 44.1 kHz or 48 kHz. 48kHz is recommended. – Removed redundant remarks.
3V0_163	7.1 - 7.3	– Modified to represent speed grade MOST150.

Glossary

Term	Definition
Boundary Descriptor	The Boundary Descriptor determines the amount of bandwidth allocated for streaming data and packet data.
Central Registry	Contains a lookup table for cross-referencing logical and functional addresses. Implemented in the NetworkMaster.
ConnectionMaster	Streaming connections are managed by a Connection Manager; the ConnectionMaster FBlock is an interface to that Connection Manager.
Connection Label	Connection Labels are used to identify streaming connections.
Control data	Data packets containing control information.
Decentral Registry	Contains a lookup table for determining available function blocks and cross -referencing logical and functional addresses.
Device	A MOST device is a physical unit, which can be connected to a MOST network via a MOST Network Interface Controller.
DeviceID	The DeviceID stands for a physical device or a group of devices in the network. The DeviceID (RxTxAdr) can represent a node position address (RxTxPos), a logical address (RxTxLog), or a group address.
FBlock	A function block. Application FBlocks group functions that are particular to a specific application, for example, radio or telephone. System FBlocks group functions that have an administrative purpose, for example, NetworkMaster or NetBlock.
FBlock Shadow	The FBlock Shadow is an implementation concept where an FBlock is controlled through a proxy for that particular FBlock.
Frame	Data transfer is organized in frames. Frames consist of administrative data and payload.
Function	A function is a part of an FBlock through which it communicates with the external world.
Init Ready event	When the system is in a Stable Lock state (i.e., operational), the Init Ready event is fired.
Method	Function that can be started and which leads to a result after a certain period of time.
NetInterface	The expression NetInterface stands for the entire communication section of a node: the physical interface, the MOST Network Interface Controller, and the Network Service.
NetOn State	The NetOn state corresponds to the “NetInterface Normal Operation” state.
NetworkMaster	The NetworkMaster controls the System State and administrates the Central Registry.
Node	A node is characterized by the existence of one NetInterface.
Notification	Notification is used to inform devices about changes in a property. All devices that are registered for that particular property in the Notification Matrix will receive a Status message with the updated value.
Notification Matrix	Devices that require status updates when a property changes can register for that particular property in the Notification Matrix.
OPType	When accessing functions, certain operations can be applied. The type of operation is specified by the OPType (e.g., Set or Get).
Packet data	Asynchronous data packets, such as IP packets.
PowerMaster	The PowerMaster is responsible for power management throughout the MOST system.
Property	Function for determining or changing the status of a device.
RxTxAdr	Either a logical node address (RxTxLog), a node position address (RxTxPos), or a group address. For receiving nodes, it is called RxAdr, for sending nodes, TxAdr.
RxTxLog	A logical node address. It must be unique in the system and is called RxLog for receiving nodes and TxLog for transmitting nodes.
RxTxPos	A node position address. The node position address is called RxPos for a receiving node and TxPos for a transmitting node. The node position address depends on the physical position of the MOST Network Interface Controller.
Source data	The term source data combines streaming data and packet data.
Streaming data	Content, such as audio or video data, which has to be transmitted synchronously or isochronously, is referred to as streaming data.
System Lock Flag	This flag indicates that the TimingMaster has detected a Stable Lock of the system.
System Scan	The process of collecting information from the NetworkSlaves, performed by the NetworkMaster.
System State	There are two System States: OK and NotOK. In OK, the system is in normal operation mode, in NotOK, it is being initialized or updated.

Term	Definition
TimingMaster	The TimingMaster provides generation and transport of the system clock.

1 Introduction

1.1 Purpose

This document, which all other MOST Specifications relate to, is the main specification of MOST (Media Oriented System Transport).

1.2 Scope

This document contains the specification of the MOST Application Layer and the MOST Network Layer.

This specification is speed grade independent. However, it is not fully backward compatible to the existing speed grades MOST25 and MOST50 as described in the MOST Specification Rev. 2.5.

1.3 MOST Document Structure

This document structure reflects the documents published by the MOST Cooperation and their internal dependencies. This structure is subject to change as new documents are published.

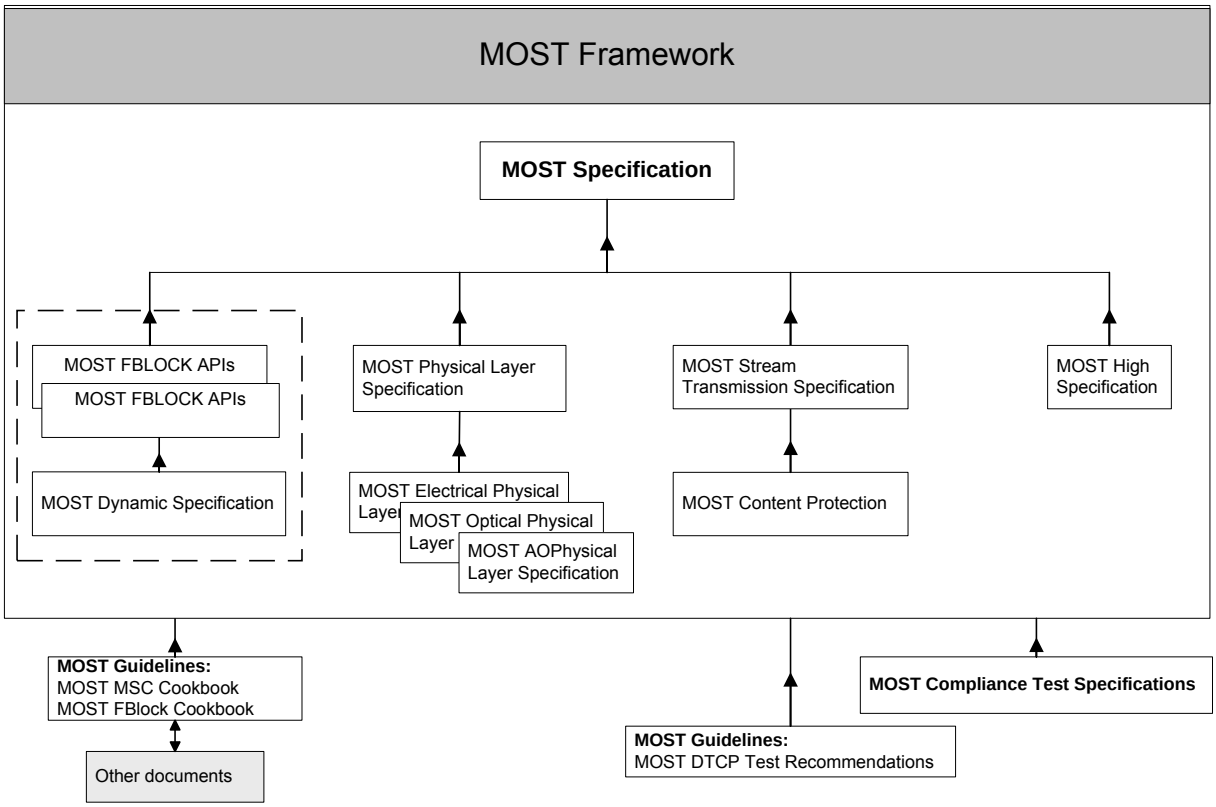

Figure 1-1: MOST document structure

The MOST Specification is a main specification within the MOST Framework. The arrows show the direction of references.

1.4 References

All documents, which are referenced by this MOST document, are listed here along with their versions.

Document	Revision
RFC 2616 Hypertext Transfer Protocol — HTTP/1.1	June 1999
The Unicode Standard	
IEEE 802.3 Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications.	2005
ISO 8859 Information technology — 8-bit single-byte coded graphic character sets	
IEC 62106:1999 Specification of the Radio Data System (RDS).	1999
ETSI EN 300401 Radio Broadcasting Systems; Digital Audio Broadcasting (DAB) to mobile, portable and fixed receivers	
Shift JIS JIS X0201:1997 and JIS X0208:1997	1997

1.5 Overview

This specification consists of two sections, namely the application section and the network section. The specification supports different physical layers, which are described in the respective documentations. In those cases when optical physical layer is mentioned in this specification, it has to be seen as an example.

2 Application Section

The Application Section comprises of Application Layer Service Specification and the Application Layer Protocol Specification. This includes access to the different data transport mechanisms, the logical device model, the functional addressing as well as the introduction of the function block (FBlock) concept.

2.1 Application Layer Service Specification

The Application Layer Service Specification defines the abstract interface between the Application and the Application Layer. To this purpose, the basic principles of MOST and its FBlock concept are introduced.

2.1.1 Basic Principles of MOST

MOST is a function oriented high-speed multimedia technology to network a variety of devices (respectively MOST nodes). MOST defines mechanisms for sending streaming data and packet-based data, and provides a complete application framework to control interaction between devices in a clearly structured way. MOST supports different speed grades and physical layers.

A MOST system consists of up to 64 nodes. MOST is a synchronous network: The TimingMaster provides the system clock with a continuous data signal. All other devices—the TimingSlaves—synchronize their operation to this base signal.
Within the synchronous base data signal, the content of multiple streaming connections and control data are transported. The Control Channel is used to initiate the streaming data connection between sender and receiver.

The bandwidth allocated for the streaming data connections is always available and reserved for the dedicated stream so there are no interruptions, collisions, or delays in the transport of the data stream. MOST is designed for high quality of service and efficient transport of audio and video.

Internet traffic or information from a navigation system is typically sent in short (asynchronous) bursts as packets and is often transported to many different places. To accommodate such signals, MOST has defined efficient mechanisms for sending asynchronous, packet based data in addition to the control data and the streaming data. These mechanisms run on top of the permanent synchronous data signal. However, the transmission of packet based data is completely separate from the Control Channel and the streaming data so that none of them interfere with each other.

In summary, MOST is a network that has mechanisms to transport all the various signals and data streams that occur in multimedia and infotainment systems.

2.1.1.1 Overview of Data Transport

For transmitting data, a MOST network provides the following types of data transport mechanisms with different characteristic properties.

2.1.1.1.1 Control Channel

On the Control Channel, data packets (for control messages) are transported to specific addresses. The Control Channel is secured by the Control Channel CRC and has an ACK/NAK mechanism with automatic retry. It is generally specified for event-oriented transmissions at low bandwidth and short packet length.

2.1.1.1.2 Streaming Data

Continuous data streams that demand high bandwidth and require time-synchronized transmission (typically multimedia data, such as audio or video) are transported using streaming data connections. The connections are administered dynamically through appropriate control messages.

Although streaming connections can be built directly between nodes that contain sources and sinks, it is recommended that the available bandwidth for streaming data connections be administered in a central manner, particularly in larger networks. Administration of the streaming resources is, in this case, handled by a Connection Manager that is responsible for all requests for establishing connections.

2.1.1.1.3 Packet Data Channel

In contrast to the Control Channel, the Packet Data Channel is specified for transmissions requiring high bandwidth in a burst-like manner. It is mainly used for transmitting data with large block size (e.g., graphics, picture formats, and navigation maps).

Just like the Control Channel, the Packet Data Channel is secured by CRC (Packet Data Channel CRC) and has an ACK/NAK mechanism with automatic retry.

2.1.2 Device Model

The following sections describe the logical model of a MOST device. A MOST device is a physical unit that can be connected to a MOST network via a MOST Network Interface Controller.

2.1.2.1 Function Block

On the application level, a MOST device contains multiple components that are called function blocks (FBlocks), for example, tuner, amplifier, or CD player. It is possible that there are multiple FBlocks in a single MOST device, such as a tuner and an amplifier combined in one case and connected to the MOST network via a common MOST Network Interface Controller.

In addition to the FBlocks, which represent applications, each MOST device has a special FBlock called the NetBlock. The NetBlock provides functions related to the entire device.

Between the FBlocks and the MOST Network Interface Controller, the Network Service forms an intermediate layer providing routines to simplify the handling of the MOST Network Interface Controller.

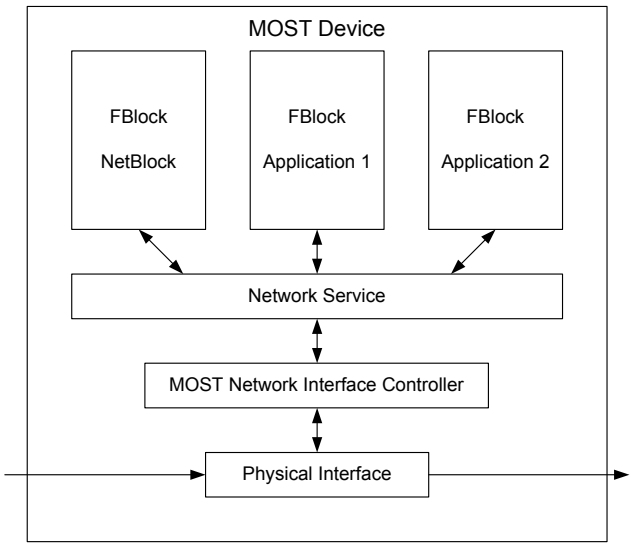

Figure 2-1: Model of a MOST device

The MOST standard supports different physical interfaces. These are described in the respective Physical Layer Specifications.

Each FBlock contains a number of single functions. For example, a CD player possesses functions such as Play, Stop, Eject, and Time Played.

2.1.2.1.1 Slave, Controller, HMI

Interaction with an FBlock requires two partners which are distributed over the MOST network: The Controller and the Slave.
The FBlock functionality resides in the Slave. The Controller sends commands to a Slave and in return receives reports from the Slave. The Slave executes the commands issued by a Controller and sends status reports to the Controller.
In a device, Controllers and Slaves can coexist. This means that a device which contains Controllers can also contain FBlocks and, therefore, be controlled by other devices. A Slave may be associated with more than one Controller. Correspondingly, a Controller may be associated with more than one Slave.

Controllers that have an interface to the user are called Human Machine Interfaces (HMIs).

Devices are commonly classified as HMI, Controller, or Slave with respect to their primary function. However, a device may contain a combination of HMI, Controller, and Slave functionality.

2.1.2.1.2 First Introduction to MOST Functions

This section gives a brief introduction to the structure of MOST functions. This knowledge is necessary to understand the following examples. Section 2.2 on page 40 explains the structure of MOST functions in more detail.

On the application level, a function is addressed independently of the device it is implemented in. Functions are grouped together in FBlocks with respect to their contents. Therefore, FBlocks are references for external applications to localize a certain function. A function is addressed in an FBlock. In order to distinguish between the different FBlocks and functions (Fkts) of a device, each function and FBlock has an identifier:

FBlockID.FktID

When accessing functions, certain operations are applied to the respective function. The type of operation is specified by the OPType. The parameters of the operation follow the OPType, resulting in the following structure:

FBlockID.FktID.OPType(Data)

2.1.2.2 Functions

A function is a defined attribute of an FBlock through which it communicates with the external world. Functions can be subdivided into two categories:

- Functions that can be started and which lead to a result after a certain period of time.
  These functions are called “methods”.
- Functions for determining or changing the status of a device, which refer to the current properties of a device.
  These functions are called “properties”.

In addition, there are events. Events result from changes of properties, if the properties are requested to report these changes (notification).

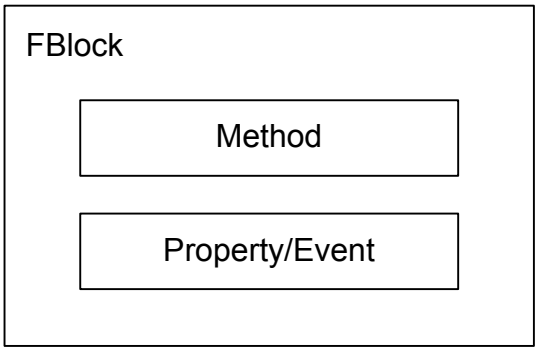

Figure 2-2: Structure of an FBlock consisting of functions classifiable as methods, properties, and events

2.1.2.3 Methods

Methods can be used to control FBlocks. In general, a method is triggered only once at a certain point of time, for example, starting the auto-scanning of a tuner. Methods can be defined without parameters or with certain parameters that specify their behavior. For example, a method “auto-scan” is possible without parameters or with a parameter that specifies the direction of the auto-scan.

When an FBlock receives a method call, it starts the respective process. If this is not possible, the FBlock has to return the respective error message to the sender of the method call. This may happen if the addressed FBlock has no method of that kind, if a wrong parameter was found, or if the current status of the FBlock prevents execution of the method.

After finishing the process, the controlled FBlock should report execution to the Controller. This report may contain results of the process, for example, a frequency found by the tuner. If a process runs for a long time, it may be useful to return intermediate results before finishing, such as informing the Controller about the successful start of the process.

For executing methods, the following kinds of messages are exchanged via the bus:

Controller	Slave
Start of a method with parameters (*StartAck*/*StartResultAck*)	Error with cause for error (*ErrorAck*) Execution report with results *(ResultAck)* Intermediate result *(ProcessingAck)*

The respective MOST mechanisms needed for this messaging are described in depth in section 2.2 on page 40.

2.1.2.4 Properties

Properties can be read (e.g., temperature), written (e.g., passwords), or read and written (e.g., desired value for speed control). For each property, the allowed operations are specified.

Within an FBlock, a property represents a value or a status.

2.1.2.4.1 Setting a Property

The process of setting a property is described by the example of the temperature setting of a heating control.

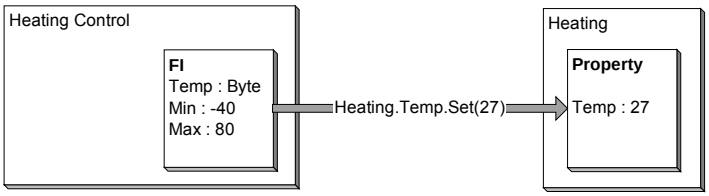

Figure 2-3: Setting a property (temperature setting of a heating)

Function Temp is a member of the FBlock Heating, so the HMI sends the instruction Heating.Temp.Set(27) to FBlock Heating.

2.1.2.4.2 Reading a Property

In order for the HMI to display the current temperature, the value of function Temp in FBlock Heating must be read. Therefore, the HMI sends the instruction Heating.Temp.Get.

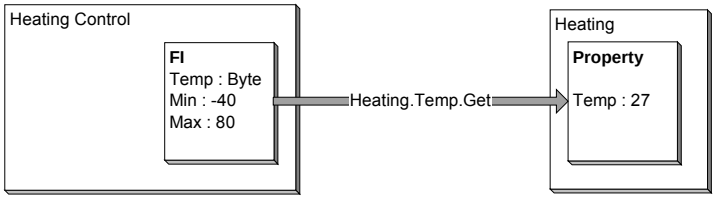

Figure 2-4: Reading a property (temperature setting of a heating)

Heating replies by sending the status message Heating.Temp.Status(27).

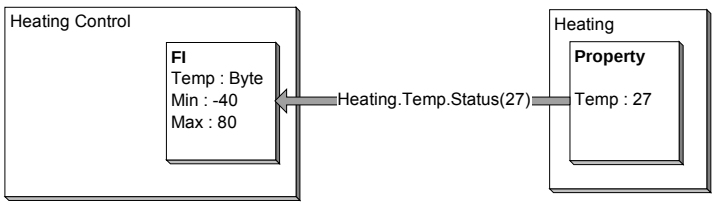

Figure 2-5: Status report of property temperature setting

For changing and reading of properties, the following types of messages are exchanged via the bus:

Controller	Slave
Setting a property (*Set*/*SetGet*) Reading a property (*Get*) Incrementing / decrementing a property (*Inc*/*Dec*)	Status of property (*Status*) Error message with cause of error (*Error*)

The MOST functions needed for this messaging are described in depth in section 2.2 on page 40.

2.1.2.5 Events

Properties of an FBlock may change without an external influence, for example, the temperature in the example above, or the current time of a CD player. To display current values using the functions described up to now, a cyclical reading of the properties (polling) would be required.

To reduce communication with FBlocks, it would be useful if FBlocks could send status reports about changes in properties without explicit requests. These are events that occur in a controlled FBlock, which initiate the sending of a report (notification).

When using notification, it is not recommended to change properties with the SetGet OPType. Use Set instead to avoid the potential transmission of two identical messages.

Notifications can be used to indicate reaching of limits, the change of measured values in FBlocks (e.g., the play time of a CD player has changed), or in the HMI (e.g., reception of a new value of mileage received via a CAN gateway). Notifications are sent only to those Controllers that have previously applied for the notification by sending an appropriate request to the Slave (and are therefore included in the Slave's Notification Matrix).
(Refer to section 2.2.5 on page 111).

2.1.2.6 Addressing MOST Functions

In a MOST network, the devices are connected in a ring structure. To address these devices, different types of addresses can be used. The MOST Network Interface Controller provides six different types of addresses, which are introduced below.

The detailed description can be found in section 3.2.2.

MOST address types:

1. **Internal Node Communication address**
   This address is reserved for internal communication in a node.

2. **Node position address (RxTxPos)**
   The node position is generated automatically in each node during the locking procedure of the MOST Network. Thus, the node position address is based on the physical position of the MOST Network Interface Controller. The node position address is 2 bytes long.
   The node position address is called RxPos for a receiving node and TxPos for a transmitting node.

3. **Logical node address (RxTxLog)**
   This address can be set by the application. It must be unique in the system and is 2 bytes long. A logical node address can be either dynamic or static. A dynamic address is calculated based on the node position address. A static address is predefined and is not recalculated on reconfiguration of the network.

   The logical node address is called RxLog for receiving nodes and TxLog for transmitting nodes.

4. **Group address**
   The group address can also be set by the application. It is 2 bytes long. A group consists of devices that have the same group address.

5. **Broadcast address**
   The broadcast address is a special group address, which is 2 bytes long. The broadcast can either be a “blocking broadcast” or an “unblocking broadcast”.

6. **Ethernet MAC address**
   This address is 6 bytes long, can be set by the application, and is static.

7. **Functional address**
   The combination of FBlockID and InstID is referred to as the functional address.

2.2 Application Layer Protocol Specification

The Application Layer protocol is the set of rules that governs the format and meaning of the information exchange between the Application Layers in various devices. The Application Layer uses the protocol to implement the Application Layer services definitions.

2.2.1 Application Messages in a MOST Network (Introduction)

The controlling mechanisms described in this document are generally independent of the kind of bus used. Messages on the application level are described in a universal way. They are transported virtually from one application to the other. In reality, they are transmitted with the help of a bus system, here the MOST network, which is described in detail below.

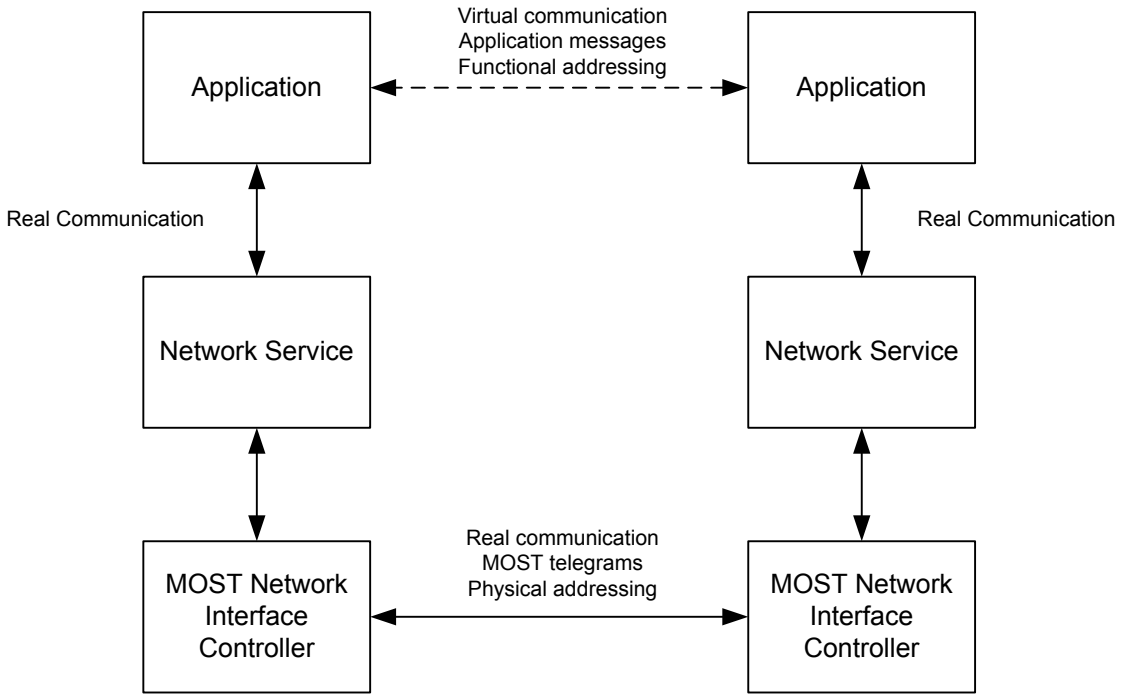

Figure 2-6: Virtual communication between two devices on the Application Layer and real communication over the network

All application messages are finally transferred via the Control Channel or the Packet Data Channel of the MOST network. From the application's point of view, all messages are passed on to the Network Service. Depending on the length, an application message is sent with a single transfer if it fits into one MOST telegram, otherwise via segmented transfer.

In a MOST Network, nodes (devices) are addressed. In order to transport a message to an FBlock, the MOST telegrams are provided with the address of the device that contains the FBlock.

Here, the entire data flow of an interaction between two devices via the Network Layer is described. One device controls the functions of the other.
The figure below shows the properties of an FBlock with the FBlockID CD and the InstID 1. The FBlock is found in the CD Changer (CDC) device.

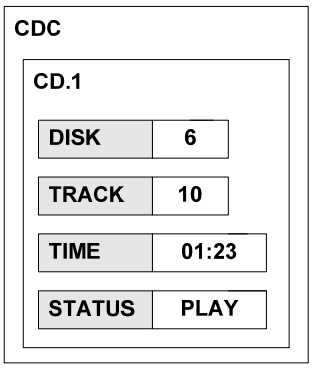

Figure 2-7: CDC device with CD Player FBlock and its functions

For example, another track can be chosen by reception of the following command:

```
CD.1.Track.SetGet(10)
```

This message is sent by a device that contains the controlling HMI. It will be passed on to the Network Service in the following form:

```
???.CD.1.Track.SetGet(10)
```

The first part is a special DeviceID, which means that the device address of the receiver is not known on the application level. The Network Service will complement the address with the address of the CD Changer. The result is:

```
CDC.CD.1.Track.SetGet(10)
```

For transmission, this is complemented by the sender's device address:

```
HMI.CDC.CD.1.Track.SetGet(10)
```

Since the receiving device knows its own device address, this address does not need to be passed on to the application level. The received message, therefore, looks like:

```
HMI.CD.1.Track.SetGet(10)
```

If the function wants to report its new status, it builds the following reply:

```
HMI.CD.1.Track.Status(10)
```

Based on this, the Network Service builds the following telegram:

```
CDC.HMI.CD.1.Track.Status(10)
```

In the HMI, the receiver's address is removed and the message is passed to the application:

```
CDC.CD.1.Track.Status(10)
```

The general data flow via the different layers in the two devices is displayed in the following figure:

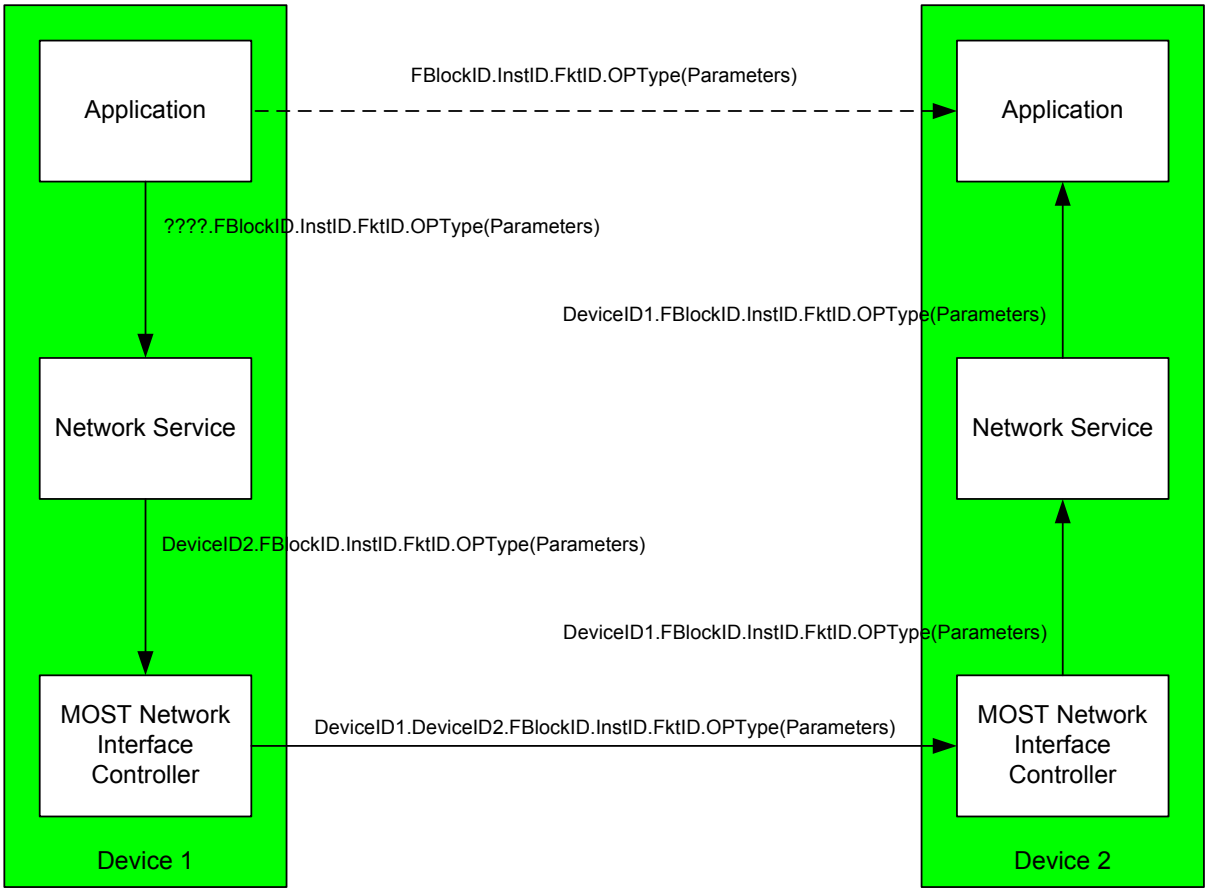

Figure 2-8: Communication between two devices via the different layers

2.2.2 Controller / Slave Communication

For communication between Controllers and Slaves, properties and methods must be differentiated. Properties describe the current operation status of the corresponding device function completely at any time. Methods are used to control a process that usually does not provide a result immediately.

2.2.2.1 Communication with Properties Using Shadows

In section “2.1.2.1.1 Slave, Controller, HMI”, a general description of the Slave/Controller concept is provided. The FBlock Shadow is an implementation concept where an FBlock is controlled through a proxy for that particular FBlock.
Below, communication between a controlling and a controlled device is explained for properties by an example:

- Controlling device (HMI device):
  Contains FBlock Shadows for controlling the CD changer and Tuner FBlocks
- Controlled device (CDC device):
  Contains only the CD changer FBlock

The properties of a device should describe the current operation status completely at any time. The figure below shows the properties of an FBlock CD changer with FBlockID CD and the InstID 1 in the CDC device.

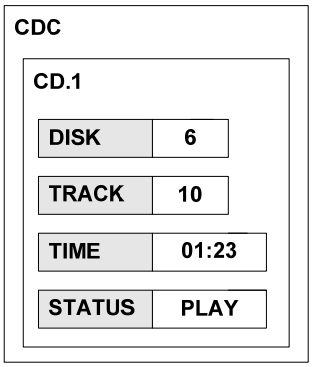

Figure 2-9: Example of a Slave device

Operation status of the player is determined by the properties Disk (number of loaded CD), Track, Time, and Status (Play, Stop, Forward, Rewind, and Eject). By changing these properties, the player can be controlled by another device.

For example, another track can be chosen by sending the following command:

```
HMI -> CDC:  CD.1.Track.Set(10)
```

If this operation is successful, the new state of the CD player is confirmed by the following reply:

```
CDC -> HMI:  CD.1.Track.Status(10)
```

Note: *The reply has the same functional address (FBlockID.InstID) as the command. Only the logical address (DeviceID) changes.*

By sending this command, the player can be stopped:

```
HMI -> CDC:  CD.1.Status.Set(Stop)
```

Also in this case, the new state of property Status can be transmitted via a message:

```
CDC -> HMI:  CD.1.Status.Status(Stop)
```

If another device is registered in the Notification Matrix of FBlock CD, these status messages are sent by the CD player even when a property changes spontaneously, for example, when the player moves to the next track during play mode.

The MOST device address of the CD changer (represented by the abbreviation CDC) together with FBlockID, InstID, and FktID describe the property to be changed. To make sure that the messages for controlling a device are transported properly, the property addressing must be unique in the entire system.

If there are multiple CD players in the system, they get different InstIDs and, in addition, different MOST addresses. Based on that, two players can be controlled by an HMI in the following way:

```
??? -> CDC1:  CD.1.Status.Set(Stop)
??? -> CDC2:  CD.2.Status.Set(Stop)
```

By this, two CD FBlocks can be addressed unambiguously, even if they are located within one physical device with one MOST address. This also guarantees that status reports can be assigned unambiguously:

```
CDC -> ???:   CD.1.Status.Status(Stop)  Status of CD in CDC

CDC1 -> ???:  CD.1.Status.Status(Stop)  Status of CD in CDC1
CDC2 -> ???:  CD.2.Status.Status(Stop)  Status of CD in CDC2

CDC -> ???:   CD.1.Status.Status(Stop)  Status of 1st Player in CDC
CDC -> ???:   CD.2.Status.Status(Stop)  Status of 2nd Player in CDC
```

The controlling device (HMI device) contains the Shadows of the functions it controls. The Shadow of a function in the HMI device represents an image of the property of the CDC device. That means, for each controlled property of the CDC device, the HMI device contains a respective variable. For the HMI device, the function seems to reside in its own memory area. This is shown in the figure below:

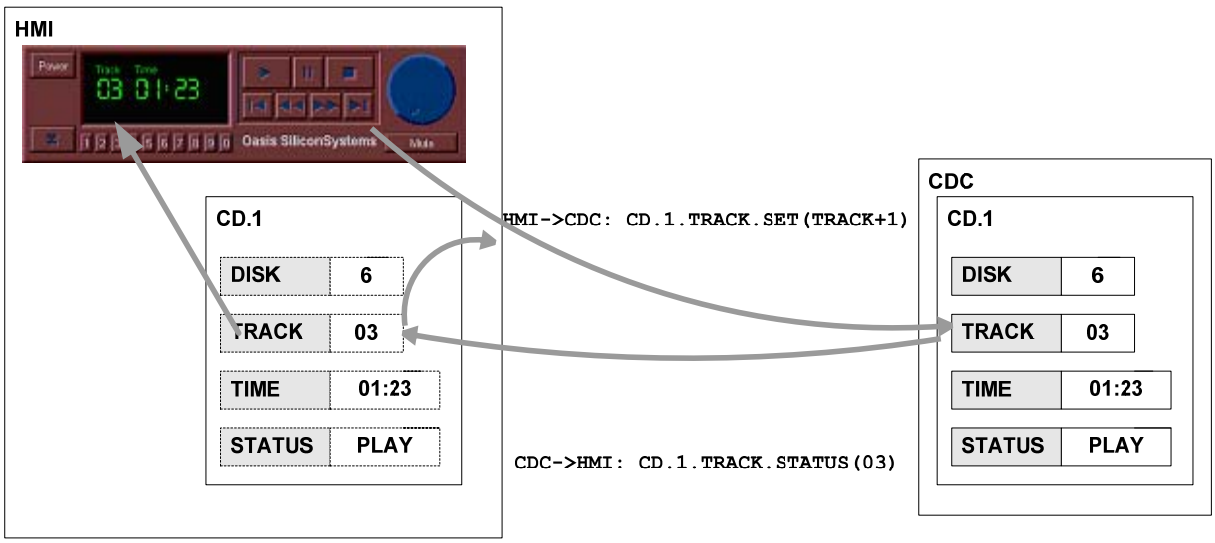

Figure 2-10: Virtual illustration of the controlled properties in the control device

The HMI device shown in Figure 2-10 has an image of the Disc, Track, Time, and Status properties of the CDC device stored in the respective. These variables are required to store the display values and can be used for control purposes, too. The example shows the flow of communication when using the “Next track” button.

When the button is clicked, the HMI device takes the contents of its local variable Track, increments it by one, and sends the command CD.1.Track.Set(Track+1) to device CDC. After the player has changed track, it replies by sending CD.1.Track.Status(3). Variable Track reacts only on that reply and stores the new value. The change of variable Track causes the HMI to update its display.

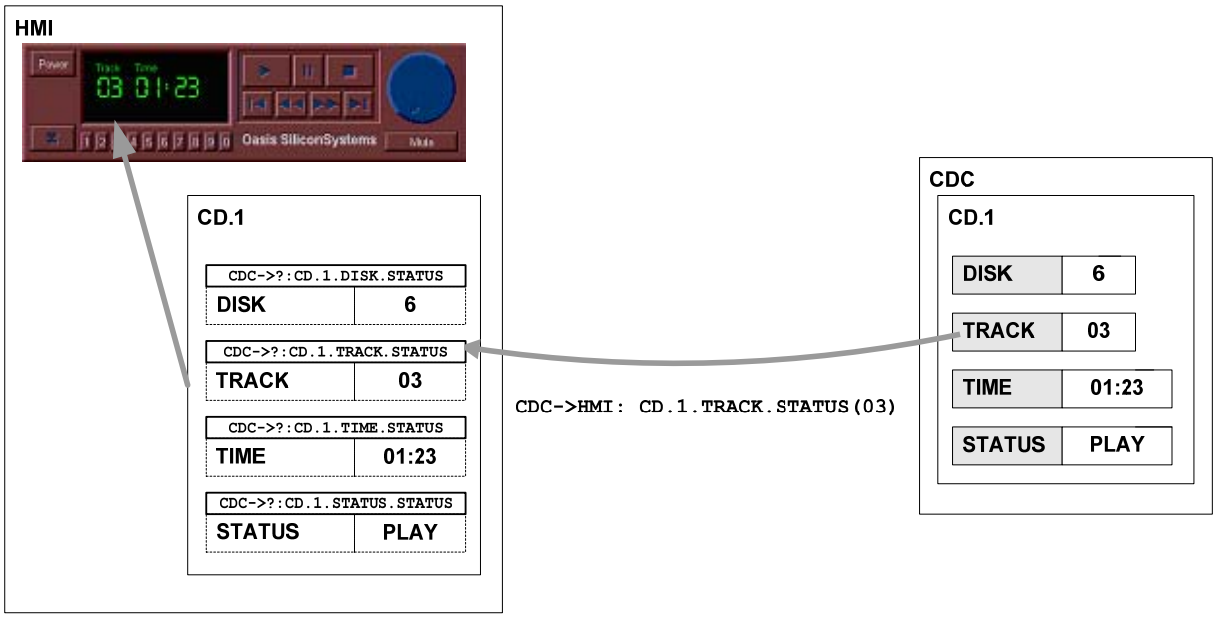

Figure 2-11: Unambiguous assignment between function and variable

The next figure shows this approach when controlling multiple devices. The HMI has an image of the controlled CD player, as well as an image of the tuner. Even during play operation of the CD player, the tuner sends status changes to the HMI. In CD operation mode, this information is not shown on the display but is stored in the respective variables. This means that the current information about the tuner is available immediately if the operation mode is changed from CD to Tuner, with no extra polling needed.

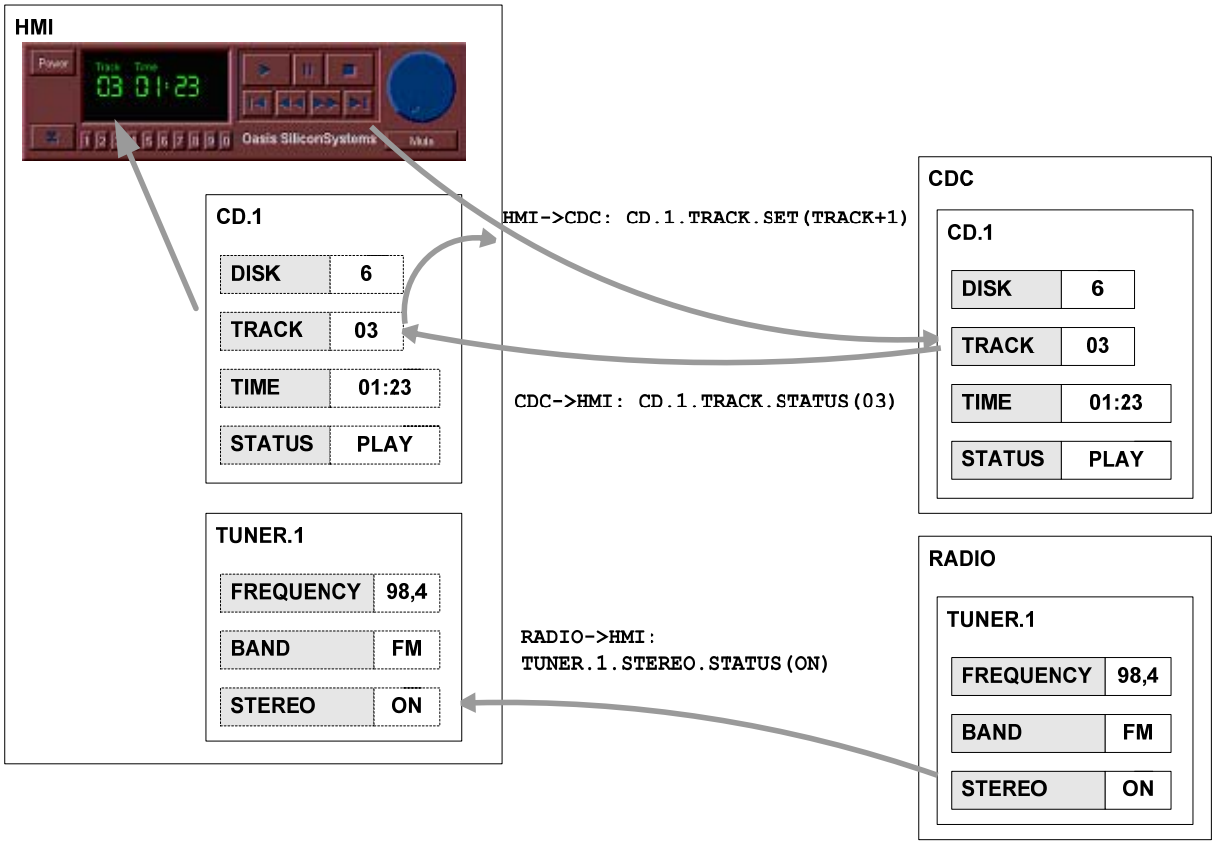

Figure 2-12: Controlling multiple devices

A similar case could be imagined for several identical CD players, where, for example, two CD players CD.1 and CD.2 can be operated via one HMI.

As shown in the following graphic, such an HMI would contain two sets of variables (Shadows), one for each CD player. The variables for CD.1 react only upon messages of CD.1, while the variables for CD.2 react only upon messages of CD.2. If both of the FBlocks are located in one device, handling would be identical.

Both sets of variables are updated, even if only one set is displayed. When switching between the players, all values are available immediately.

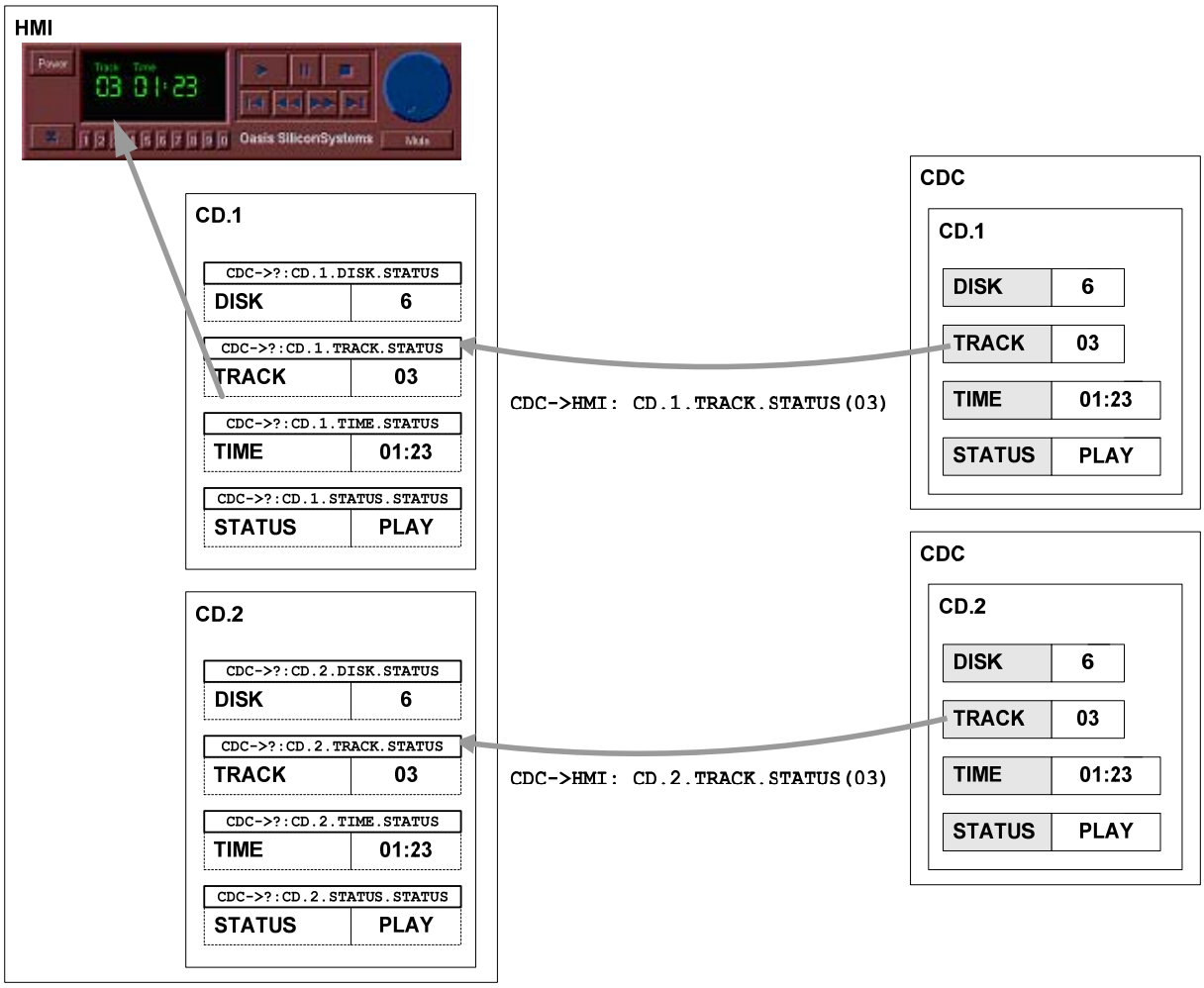

Figure 2-13: Controlling two identical devices

2.2.2.2 Communication with Methods

The main difference between methods and properties is that methods are used to control a process that usually does not provide a result immediately, while a property always has a current status.

Methods often interact with multiple Controllers and therefore may have different states at one time:
It may happen that a method is processing a request for one Controller, while it appears to another Controller to be busy. In addition, in methods, a process is triggered that has a longer processing time. The Controller may need to wait for a result.

2.2.2.2.1 Using Methods with SenderHandle

As several tasks within a device might access one method at the same time, it must be possible to route the answer back to the respective task. This is achieved by using the SenderHandle parameter.

On Slave side, the SenderHandle in combination with DeviceID (of the originator) and FBlockID.InstID is used to identify a method under execution. On the Controller side, the SenderHandle is used to dispatch the responses of the Slave. It is in the responsibility of the controlling device to choose an unambiguous value for SenderHandle. The SenderHandle must not contain any application relevant information.

The presence of the SenderHandle is indicated by using the Ack-variants of the OPTypes (e.g., StartResultAck, ResultAck, and ErrorAck) (see 2.2.3.5.2).
An example:

```
Controller -> Slave: FBlockID.InstID.StartResultAck   (SenderHandle, Data)

Slave -> Controller: FBlockID.InstID.ResultAck        (SenderHandle, Data)

Slave -> Controller: FBlockID.InstID.ErrorAck         (SenderHandle,
                                                        ErrorCode, ErrorInfo)
```

One example for the use of SenderHandles is the SMS service in a GSM module of a telephone device: in the HMI, three tasks are attempting to send SMS text messages independently from each other. The message of task 1 is successfully sent, the message of task 2 is buffered, while the message of task 3 is rejected.
The respective status messages must now be assigned, which requires an approach that is described below.

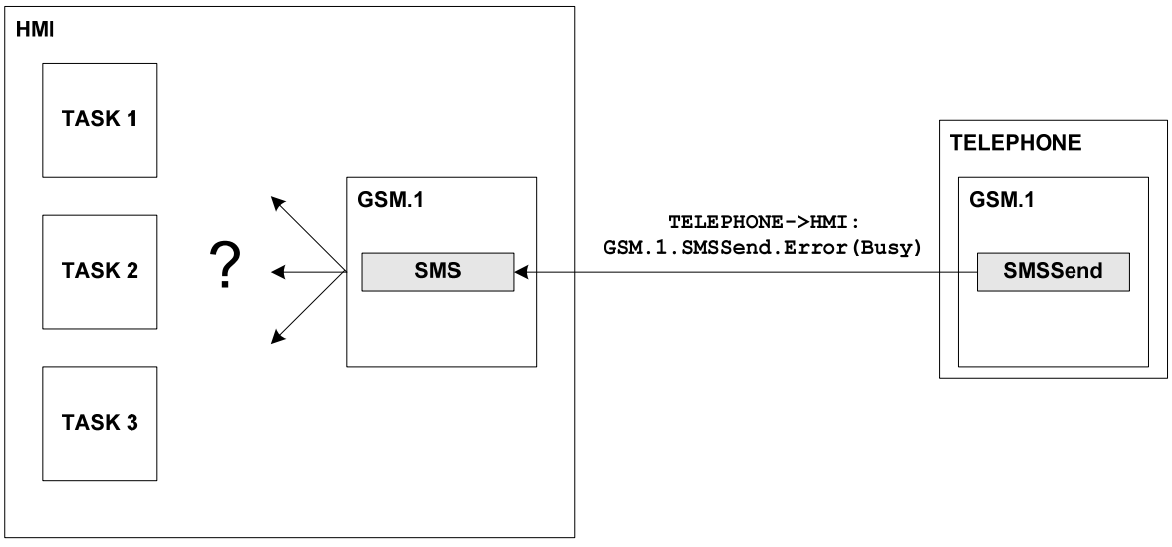

Figure 2-14: Routing answers in case of multiple tasks (in one Controller) using one function

The SenderHandle is set by the HMI in StartResultAck. The SenderHandle will not be interpreted by the telephone, but will be returned in an answer (ProcessingAck, ResultAck or ErrorAck).

The SMS call in Task 1 may look like:

```
HMI -> Telephone: Telephone.1.SMSSend.StartResultAck (SenderHandle1, SMSData)
```

After successful transmission, Task 1 receives:

```
Telephone -> HMI: Telephone.1.SMSSend.ResultAck (SenderHandle1)
```

If Task 3 attempts to send in the meantime, it transmits:

```
HMI -> Telephone: Telephone.1.SMSSend.StartResultAck (SenderHandle3, SMSData)
```

It then receives this answer:

```
Telephone -> HMI: Telephone.1.SMSSend.ErrorAck (SenderHandle3,
                                                ErrorCode="Busy")
```

2.2.2.2.2 Using Methods without SenderHandle

In special cases, communication with methods is equal to communication with properties. This means that a Controller controls a function in a Slave device and there will be a reply to the controlling device.

The behavior of the non-Ack OPTypes (e.g., StartResult, Result, and Error) is identical to that of the Ack-variants. The only difference is that the SenderHandle parameter is omitted.

Note: *Methods without SenderHandle are deprecated.*

2.2.3 Structure of MOST Messages

The principal structure of messages on the Application Layer is:

DeviceID . FBlockID . InstID . FktID . OPType (Data)

Compared to section 2.1.2.1.2 on page 35, two components were added: DeviceID and InstID. The individual elements are explained below.

2.2.3.1 DeviceID

The DeviceID stands for a physical device or a group of devices in the network (ID is network specific and has a length of 16 bits). It precedes the message and does not need to be interpreted on the application level.

If a function receives a message, the DeviceID contains the logical node address of the sender. In case of an answer, it precedes the message as the receiver's address.

If the controlling device does not know the Slave device's address, the DeviceID is set to 0xFFFF. In that case, it is corrected by the Network Service of the sender. This mechanism is not available for communication directed from a Slave device to a controlling device.

2.2.3.1.1 Basics for Automatic Adding of Device Addresses

Since applications know only functional but not device addresses, a message that is transported must be complemented by the device address (DeviceID). There are two possible ways to achieve this:

1. When the Slave answers a request it already has the DeviceID of the target node because it was reported during the request.
2. When the Controller is sending a message and does not know the DeviceID of the target, it sets the DeviceID to 0xFFFF. The ID is complemented by the Network Service and inserted into the MOST telegram as TxAdr. Refer also to sections 3.1.3.3.2 and 3.1.3.4.1 for information regarding the Central Registry and the Decentral Registries respectively.

This flow chart describes how a device seeks the logical node address of a communication partner:

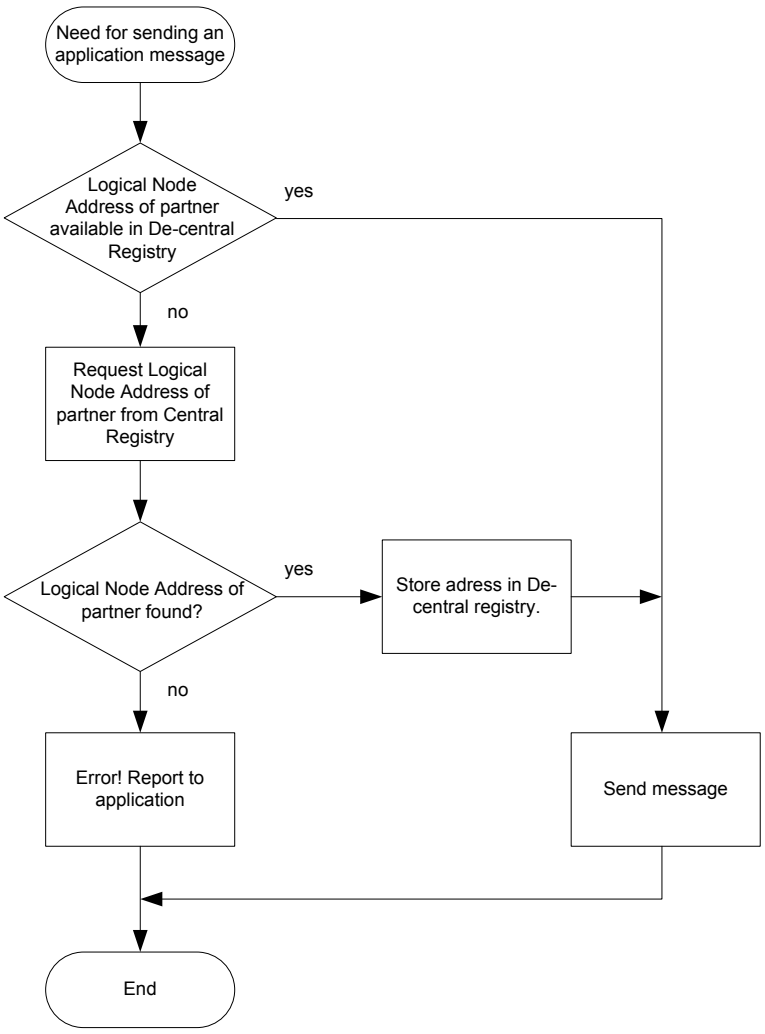

Figure 2-15: Seeking the logical address of a communication partner

2.2.3.2 FBlockID

The FBlockID is the identifier of a special FBlock. Every FBlock with a certain FBlockID must contain certain specific functions. In addition to those mandatory functions, it may contain other optional functions.

“System Specific” proprietary FBlockIDs can be used by a System Integrator (e.g., a car maker). They are specific for a system and are coordinated by the System Integrator between the suppliers developing devices for this system. A second kind of proprietary FBlockIDs are called “Supplier Specific”. Those FBlockIDs can be used by suppliers for any proprietary purpose. The special FBlockID 0xFF addresses all FBlocks within a MOST device, except the NetBlock. Since this can be regarded as a broadcast function, error status messages must not be returned.

The table below shows a collection (incomplete) of FBlockIDs:

Kind	FBlockID 8 Bit	Name	Explanation
Administration	**0x0x**		
	0x00	—	This FBlockID is used for communication between Network Service and Network Interface Controller. Telegrams that are related to network tasks are sent and received here. They are not passed to the application.
	0x01	NetBlock	Mandatory for each device
	0x02	NetworkMaster	Mandatory for each system
	0x03	ConnectionMaster	
	0x04	PowerMaster	
	0x05	Vehicle	
	0x06	Diagnosis	
	0x08	Router	
	0x09	DebugMessages	
	0x0E	Tool	
	0x0F	EnhancedTestability	Mandatory for each device
Operation	**0x1x**		
	0x10	Human Machine Interface (HMI)	
	0x11	Speech Recognition	
	0x12	Speech Output Device	
	0x13	Speech Database Device	
Audio	**0x2x**		
	0x20	Audio Master	
	0x21	Audio DSP	
	0x22	Audio Amplifier	
	0x23	HeadphoneAmplifier	
	0x24	AuxiliaryInput	
	0x25	AuxiliaryOutput	
	0x26	MicrophoneInput	
	0x28	Handsfree Processor	
	0x29	AuxiliaryInputOutput	
Drives	**0x3x**		
	0x30	Audio Tape Recorder	
	0x31	Audio Disk Player	
	0x32	ROM Disk Player	
	0x33	Multimedia Disk Player	
	0x34	DVD Video Player	

Table 2-1: FBlockIDs (part 1)

Note: *EnhancedTestability (0x0F) is mandatory for each device due to compliance reasons.*

Kind	FBlockID 8 Bit	Name	Explanation
Receiver	**0x4x**		
	0x40	AM/FM Tuner	
	0x41	TMCTuner	
	0x42	TV Tuner	
	0x43	DAB Tuner	
	0x44	Satellite Radio	
	0x45	TPEG Tuner	
	0x46	ESDR	ETSI Satellite Digital Radio
Communication	**0x5x**		
	0x50	Telephone	
	0x51	Phonebook	
	0x52	Navigation System	
	0x53	TMC Decoder	
	0x54	Bluetooth	
Video	**0x6x**		
	0x60	Display	
	0x61	Camera	
	0x62	Video Tape Recorder	
Reserved	**0x70 ...0x9F**	**Reserved for future use**	
Proprietary			
	0xA0 …0xC7	**System Specific**	
	0xC8	**Reserved for Compliance Testing**	
	0xC9 …0xEF	**System Specific**	
	0xF0 …0xFB	**Supplier Specific**	
	0xFC	**Reserved**	
	0xFD…0xFE	**Supplier Specific**	
	0xFF	**All**	

Table 2-2: FBlockIDs (part 2)

The range between 0x00 and 0x9F can only be assigned by the MOST Cooperation.

2.2.3.2.1 Handling of Supplier Specific FBlocks

Supplier specific FBlocks are not reported in NetBlock.FBlockIDs.Status.
Supplier specific FBlocks have to ignore all commands which contain FBlockID 0xFF.

2.2.3.3 InstID

There may be several equal[1] FBlocks (Instances) with the same FBlockID in the system (two CD changers, four active speakers, several diagnosis blocks, etc.). In order to address these FBlocks unambiguously, the FBlockID is complemented by an eight-bit instance identification number (InstID). The combination of FBlockID and InstID is referred to as the functional address.

2.2.3.3.1 Uniqueness of Functional Addresses

Each device is responsible for the uniqueness of its functional addresses within the device. The NetworkMaster is responsible for the uniqueness of functional addresses within the entire system. Refer to section 3.1.3.3.

2.2.3.3.2 Assigning InstID

By default, every FBlock has InstID 0x01. In case there are several FBlocks of the same kind within one MOST device, the default numbering within the device starts at 0x01 and is then incremented. In principle, as long as the InstID provides the possibility to differentiate between equal FBlocks, the InstID can be chosen freely. For example, in static systems the System Integrator may choose to use hard coded InstIDs or set the InstIDs depending on certain ranges with respect to the supported functions of the FBlock.

Note: *Wildcards must not be used for InstID assignment.*

2.2.3.3.3 InstID of NetBlock

InstIDs of NetBlocks are derived from the node position address of the MOST devices. Therefore, they start counting at 0x00.

2.2.3.3.4 InstID of NetworkMaster

The InstID of the NetworkMaster may be zero; default value is 0x01. Requests to NetworkMaster shall be sent to InstID 0x00 (wildcard ref 2.2.3.3.6).

2.2.3.3.5 InstID of FBlock EnhancedTestability

InstIDs of FBlock EnhancedTestability are derived from the node position address of the MOST device. Therefore, they start counting at 0x00.

[1] The expression “equal” means that those function blocks have the same functionality (e.g., two CD drives). This means that the basic functions are equal, but there is the possibility that they differ with respect to the total functionality (e.g., CD drive with or without random play).

2.2.3.3.6 InstID Wildcards

There are some special InstID values (wildcards) that can be used when addressing FBlocks. They will be treated as follows:

0x00	Don't care (within a device). The device dispatches the message to one specific FBlock in the device.
0xFF	Broadcast (within a device). The message is dispatched to all instances of the matching FBlock.

When replying to a request, wildcards may be used only under certain error conditions (see a-b below). In all other cases, all replies shall use the correct InstID of the respective FBlock.

a) When any FBlock is addressed by using InstID 0x00 or 0xFF, but the FBlock is not actually implemented, the returned error message ("FBlockID not available", ErrorCode 0x01) has to contain the same InstID as used in the original command message (0x00 or 0xFF respectively).

b) The error message "Segmentation Error" (ErrorCode 0x0C) shall have the same InstID as the original message. This error is handled on the transport layer but not on the Application Layer, therefore, the InstID is not evaluated.

2.2.3.4 FktID

The FktID represents a function. This means a function unit (Object) within a device that provides operations that can be called via the network. Examples for functions are: Play status of a drive, speed limit in an on-board computer, etc. On the network level, the FktID is encoded in 12 bits, so 4096 different methods and properties can be encoded per FBlock. On the application level, the FktID is extended to 2 bytes. Exceptions to this rule will be explicitly marked.

The address range of FktIDs is subdivided in the following sections:

1. **Coordination (0x000...0x1FF)**
   Functions for administrative purposes in an FBlock.

2. **Application (0x200...0x9FF)**
   Functions in the Application range represent the main functionality of an FBlock. Depending on the FBlock range, they are defined either by the MOST Cooperation, the System Integrator, or the Supplier.

3. **Unique (0xA00...0xBFF)**
   Functions that are defined unambiguously in the entire system.
   Attention, these must be coordinated throughout the entire system!

4. **Proprietary / System Specific (0xC00...0xEFF)**
   Functions that can be used by any System Integrator (car maker). They are specific to a system and are coordinated by the System Integrator between the suppliers developing devices for this system.

5. **Proprietary / Supplier Specific (0xF00...0xFFE)**
   Functions that can be used by suppliers for any proprietary purpose.

   - Supplier specific functions are not reported in <FBlockID>.FktIDs.Status.
   - If a supplier specific property supports notification, notification of this property is not affected by the <FBlockID>.Notification.Set(SetAll) command.
   - If a supplier specific property supports notification, notification of this property is cleared by the <FBlockID>.Notification.Set(ClearAll) command.

Some FktIDs in an FBlock that contains an application are predefined:

0x000	**FktIDs**	Reports the FktIDs of all functions contained in the FBlock (refer to section 3.1.3.4.6 on page 156).
0x001	**Notification**	Distribution list for events (refer to section 2.2.5 on page 111).
0x002	**NotificationCheck**	Check whether the distribution list for events is still as it should be.

When developing proprietary FBlocks, all possible Function IDs can be used freely, except those taken from the ranges:

- **Unique**
- **Coordination**

In case proprietary FBlocks contain functions within the ranges Unique or Coordination, those functions must be in accordance with MOST FBlock Specifications.

For each function it can be determined if the use of function is mandatory, optional, or depends on a condition specific to the application domain.

The following table regulates who is authorized to use certain FBlockID/FktID combinations.

FktID / FBlockID	Coordination (0x000…0x1FF)	Application (0x200…0x9FF)	Unique (0xA00…0xBFF)	Proprietary/ System Specific (0xC00…0xEFF)	Proprietary/ Supplier Specific (0xF00…0xFFE)
MOST Co. (0x00…0x9F)	MOST Co.	MOST Co.	MOST Co.	System Integrator	Supplier
System Specific (0xA0…0xC7, 0xC9…0xEF)	MOST Co.	System Integrator	MOST Co.	System Integrator	Supplier
Supplier Specific (0xF0…0xFB, 0xFD…0xFE)	MOST Co.	Supplier	MOST Co.	Supplier	Supplier

Table 2-3: Responsibilities for FBlockID and FktID ranges

Note: *It is recommended that before using any proprietary function or proprietary FBlock, a Controller verifies the identity of the device. This can be done, for example, by reading the DeviceInfo or FBlockInfo property.*

2.2.3.5 OPType

The OPType indicates which operation must be applied to the property or method specified in FktID:

OPType	For Properties	For Methods
Commands:		
0	Set	Start[1]
1	Get	Abort[1]
2	SetGet	StartResult[1]
3	Increment	Reserved
4	Decrement	Reserved
5	GetInterface[2]	GetInterface[2]
6	Not allowed	StartResultAck
7	Not allowed	AbortAck
8	Not allowed	StartAck
Reports:		
9	ErrorAck[3]	ErrorAck
A	Not allowed	ProcessingAck
B	Reserved	Processing[1]
C	Status	Result[1]
D	Not allowed	ResultAck
E	Interface[2]	Interface[2]
F	Error	Error[4]

Table 2-4: OPTypes for properties and methods

[1] These non-Ack OPTypes for Methods are deprecated. They lack a SenderHandle parameter.
[2] GetInterface and Interface are optional OPTypes.
[3] ErrorAck is required for properties to report syntax errors in conjunction with illegal OPTypes (ErrorCode 0x01...0x04).
[4] OPType Error may be used in conjunction with methods in cases where the SenderHandle parameter is not known or not applicable.

2.2.3.5.1 Error

Error is reported only to the Controller that has sent the instruction. On Error, an error code is reported in the data field (Data[0]), along with additional information as shown in Table 2-5 and Table 2-6.

Note: *Data[0]...Data[n] represent the payload of an application message; see 3.2.5.2 Application Message Service (AMS) for details.*

ErrorCode Data[0] on ErrorAck Data[2][1]	ErrorCode Description	ErrorInfo Data[1]...Data[n] on ErrorAck Data[3]...Data[n]	ErrorInfo Description
Syntax Errors			
0x01	**FBlockID not available**	—	No Info
0x02	**InstID not available**	—	No Info
0x03	**FktID not available**	—	No Info
0x04	**OPType not available**	Return OPType	Invalid OPType
Segmentation Error			
0x0C	**Segmentation Error** After this ErrorCode, the following ErrorInfo 0x01 up to 0x07 can be sent.	0x01	First segment missing, that is, the first telegram of a segmented message was not received.
		0x02	Target device does not provide enough buffers to handle a message of this size.
		0x03	Unexpected segment number.
		0x04	Too many unfinished segmentation messages pending.
		0x05	Timeout while waiting for next segment.
		0x06	Device not capable to handle segmented messages.
		0x07	Segmented message has not been finished before the arrival of another message with identical FBlockID, InstID, FktID, and OPType sent by the same node.
		0x08	Reserved, must not be used.
Application Errors			
0x05	**Invalid length**	—	No Info
0x06	**Parameter wrong / out of range** One or more of the parameters were wrong, i.e., not within the boundaries specified for the function. Example: Function Temp shall be set to 200, although maximum value is 80.	Return Parameter	Unsigned Byte containing the position of the parameter, where 1 corresponds to the first parameter. Value of first incorrect parameter only (optional).[2]
0x07	**Parameter not available** One or more of the parameters were within the boundaries specified for the function, but are not available at that time.	Return Parameter	Unsigned Byte containing the position of the parameter, where 1 corresponds to the first parameter. Value of first unavailable parameter only (optional).[2]
0x08	**Reserved. Usage deprecated**	—	No Info
0x09	**Reserved. Usage deprecated**	—	No Info
0x0A	**Reserved**	—	No Info
0x0B	**Device Malfunction**	—	No Info

Table 2-5: ErrorCodes and additional information (part 1)

[1] ErrorAck requires a SenderHandle in Data[0] and Data[1]. This results in different ErrorCode and ErrorInfo positions for Error and ErrorAck.
[2] Parameters with higher position numbers have been ignored.

ErrorCode Data[0] on ErrorAck Data[2]	ErrorCode Description	ErrorInfo Data[1]...Data[n] on ErrorAck Data[3]...Data[n]	ErrorInfo Description
0x20	**Function specific** After this ErrorCode, any function specific ErrorInfo can be sent.	0x01...0xBF	Function specific ErrorInfo.
		0xC0…0xEF	System Integrator specific
		0xF0…0xFE	Supplier specific
0x40	**Busy** Function is available, but is busy	—	No Info
0x41	**Not available** Function is implemented in principle, but is not available at the moment	—	No Info
0x42	**Processing Error**	—	No Info
0x43	**Method Aborted** This ErrorCode can be used to indicate, that a method has been aborted by the Abort/AbortAck OPTypes	—	No Info
System Integrator Specific Errors			
0xC0...0xEF	**System Integrator (e.g., car maker) specific.**	Optional	
Supplier Specific Errors			
0xF0...0xFE	**Supplier specific** After this ErrorCode, any supplier specific ErrorInfo can be sent.	Optional	Supplier specific ErrorInfo.

Table 2-6: ErrorCodes and additional information (part 2)

Controllers shall react to errors in a way that guarantees the best overall system stability, that is, the devices should be designed as fault tolerant systems. With respect to that, the Slaves also shall rely on the error messages[1] defined in the ErrorCodes table. ErrorCodes in the ranges 0x01…0x0B and 0x40…0x43 shall be preferred over the definition and use of function specific ErrorCodes (0x20).

The following requirements apply to all errors except segmentation error:
For avoiding infinite loops with respect to reporting errors, errors are reported only from Slave to Controller. In addition, no reply of error messages is allowed on reception of multicast messages (broadcast, groupcast, as well as wildcards for “all FBlocks”— see 2.2.3.2— and “all instances” — see 2.2.3.3.6).

By OPType Error, different kinds of errors are reported. Incoming messages are scanned for all these errors:

1. Syntax Error (ErrorCode 0x01…0x04)
A syntax error occurs, if, for example, a function is accessed that does not exist or if an OPType that is not implemented is called. Syntax errors are reported by the ErrorCodes 0x01…0x04. A syntax error will be reported directly after reception of a faulty command. This also applies to methods, which will not be started in that case. A Slave must report ErrorCode 0x01 if an unavailable FBlockID was requested. ErrorCode 0x02 must be reported if the requested InstID is not available.
Example for requesting a non-existing FBlock:

```
SrcAdr -> TrgAdr:
FBlockID.InstID.FktID.OPType(...)
//if FBlock not available:
TrgAdr -> SrcAdr:
FBlockID.InstID.FktID.Error(ErrorCode = 0x01)
```

[1] Error messages in general are not limited to single telegrams (except segmentation errors).

2. Application Error – Parameter Error (ErrorCode 0x05…0x06)
The specified length does not match the actual length of the data field. There have been too few parameters, too many parameters, or one parameter is out of range. Parameter errors are reported by the ErrorCodes 0x05 and 0x06. Messages are only accepted when being completely correct. In particular, this means that the length of the parameter area must be correct. The only exception is the handling of Arrays that are too short (refer to section 2.2.4.2 on page 86).

3. Application Error – Temporarily not Available (ErrorCode 0x07, 0x40, 0x41)
In some cases it may happen that the message is correct but the execution is not possible at the moment. The following distinction of cases must be performed:

- It may be that both methods and properties are implemented but cannot be executed due to operation status. An example of a method would be SMSSend of the telephone, which cannot be executed if the communication network is not available. In case of being called anyhow, it would report an OPType Error with ErrorCode 0x41 “not available”. In such a case, the application can supervise the status of the telephone and may repeat the sending of the SMS as soon as the network is available again.

- A method can be available but may be busy at the moment. So it would be possible that method SMSSend of the telephone is busy in sending another SMS. In that case an ErrorCode 0x40 “busy” would be reported. Here, the application may perform retries. This case can only occur in connection with methods.

- A property cannot be busy by definition. It is solely possible that a value is within the valid range but is not selectable at the moment. An example can be property DeckStatus of the CD drive, which cannot be set to “Play” if there is no CD loaded. Depending on the system design, this would generate an ErrorCode 0x07 “parameter not available”.

4. Application Error – General Execution Error (ErrorCode 0x42)
Especially when using methods, execution errors may occur. In general, such an error (unspecific; command was correct, but execution failed) may be reported by ErrorCode 0x42 “processing error”.

5. Application Error – Specific Execution Error (ErrorCode 0x20)
Besides the already listed errors, a MOST application may report specific errors during execution by using OPType Error as well. Here, ErrorCode 0x20 “function specific” is used.

6. Application Error – Device Malfunction (ErrorCode 0x0B)
This error indicates that the requested function is temporarily not available due to a device malfunction.

7. Segmentation Error (ErrorCode 0x0C)
Errors during reassembling the original message in the receiver can be caused, for example, by missing segments, wrong order of arrival, or exceeding the timeout between two segments. In case of such an error, the parts of the message that have already been received are discarded. In addition, the application within the receiver is notified of the error by the Network Service.

Note: *“Segmentation Error” is not limited to the context of segmented messages. The error might also be reported as a result of a single transfer reception failure. The most likely reason in that case is an input buffer overflow.*

The segmentation error notifies the sender about the failure of the transfer. Therefore, the sender’s application may react in an appropriate way, for example, by trying to send the same message again. The reaction depends on the respective problem that caused the error.

Unlike all other errors, segmentation errors are also reported from Controller to Slave.

Since the segment containing the sender handle in case of an Ack method may be missing, Segmentation Error is never sent as an ErrorAck message.

8. Application Error – Method Aborted (ErrorCode 0x43)

This error is used in case of abortion of methods by OPType Abort or AbortAck. A MOST device called "A" starts a method in device "B". Due to some exceptional events, a third device "C" aborts the method running in device "B". In that case, device "B" reports error "Method Aborted" to both the device that started the method and to the device that aborted it since they are both currently involved in the process. No other Controllers need to know about this.
The error report "Method Aborted" must always be sent back as a result of a successful abort request, even if the method is not executed anymore, for example, because it has finished successfully already.

The examination and processing of errors is done in the logical and temporary sequence as described above and in Figure 2-16 on page 62.

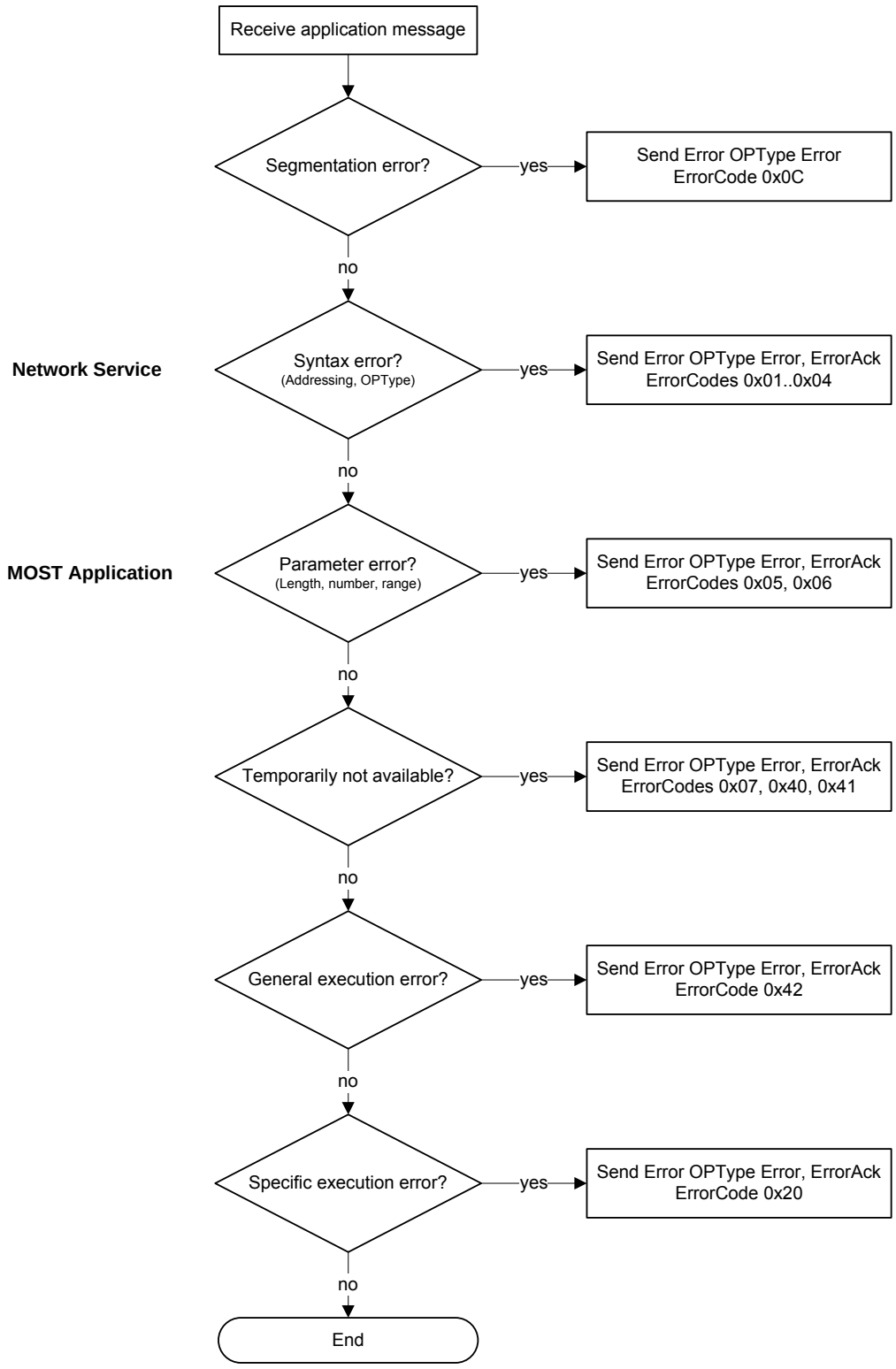

Figure 2-16: Processing of messages including error check on different layers

Higher levels of error management and individual error messages are to be specified individually.

2.2.3.5.2 StartAck, StartResultAck, ProcessingAck, ResultAck, ErrorAck

The behavior is equal to that of Start, StartResult, Processing, Result, and Error (refer to section 2.2.3.5.4 on page 65). The only difference is that the first parameter transports the SenderHandle (refer to section 2.2.2.2 on page 48).

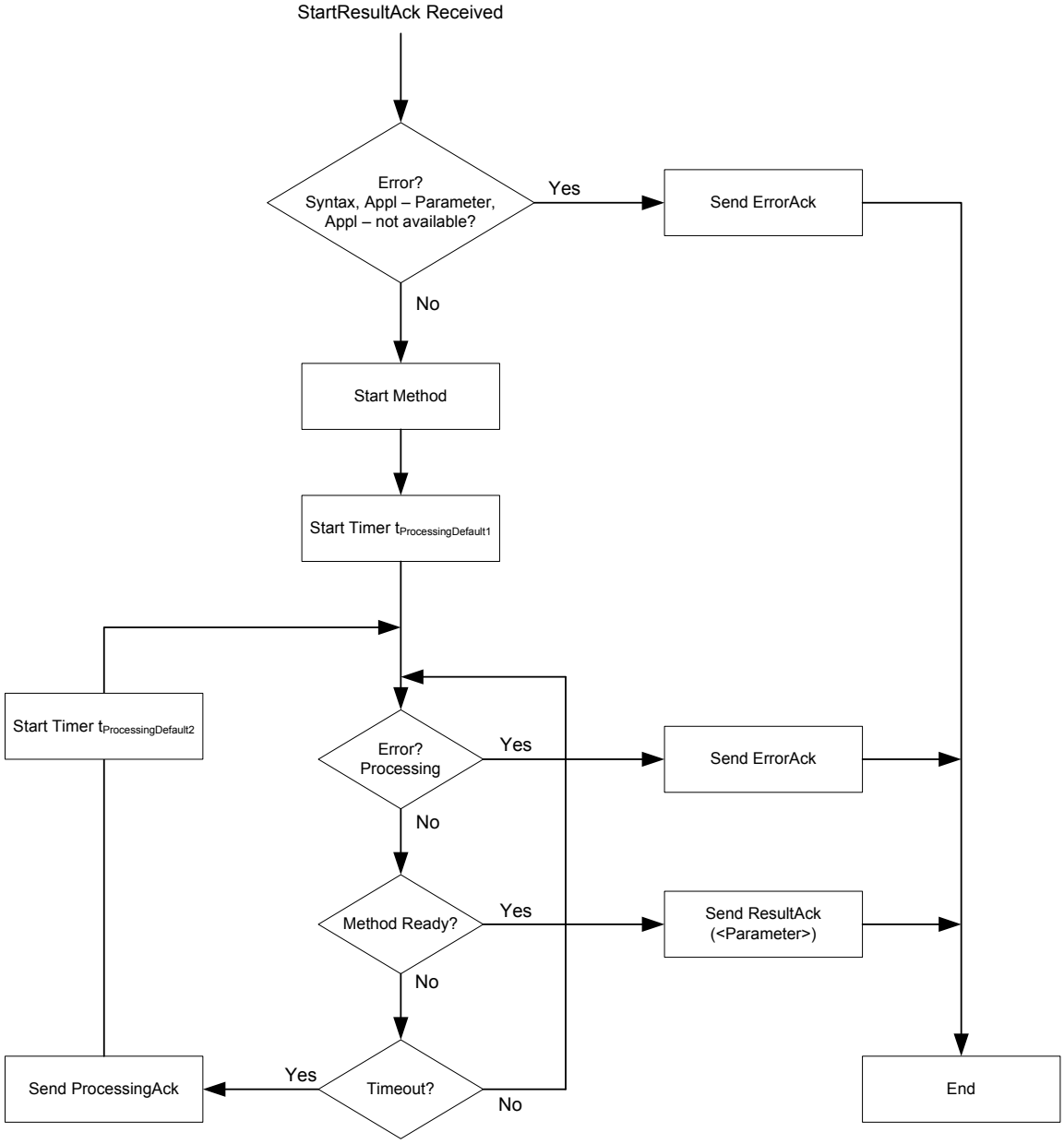

Figure 2-17: Flow for handling communication of methods (Slave's side)

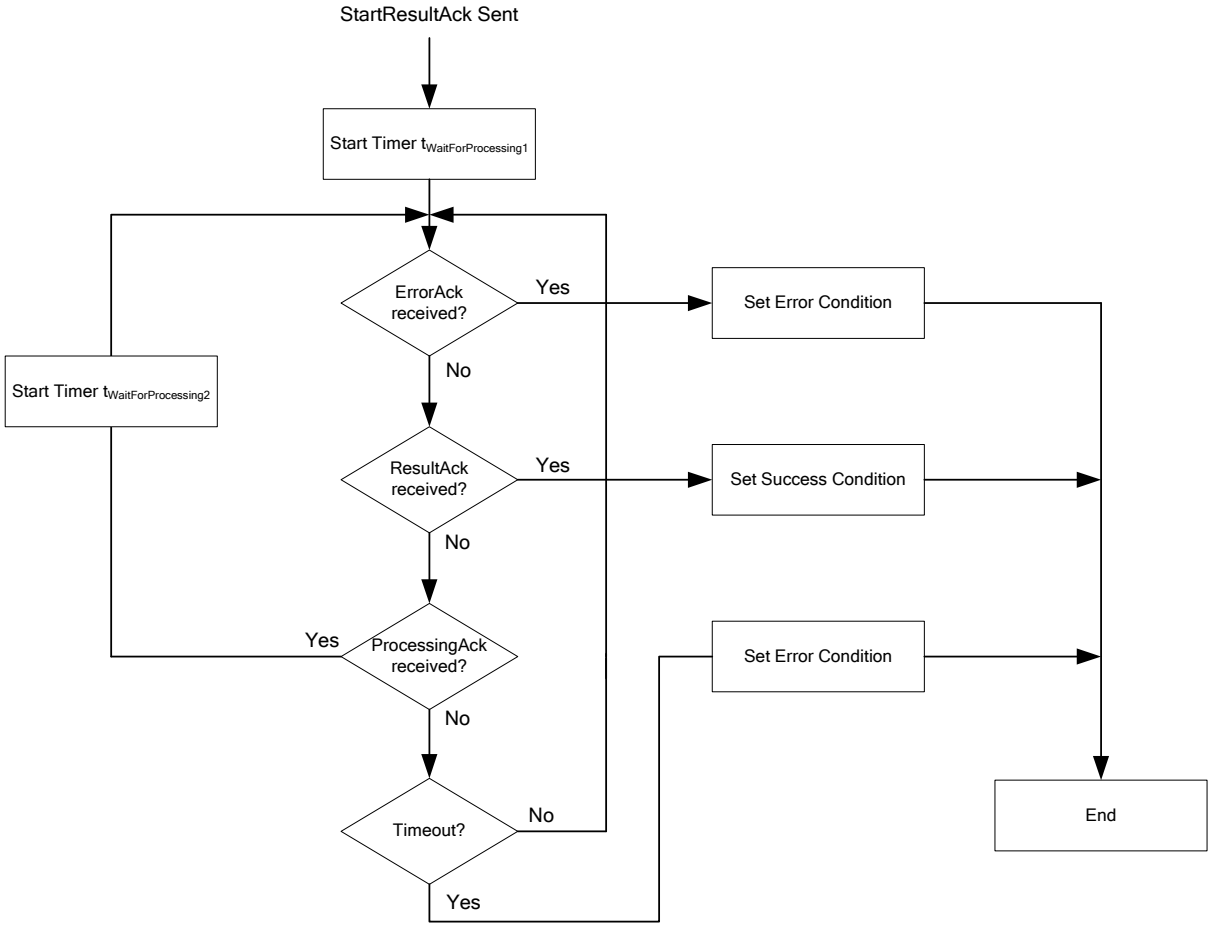

Figure 2-18: Flow for handling communication of methods (Controller's side)

2.2.3.5.3 Start, Error

By using Start, a Controller triggers a method. This approach is useful only for methods that do not return results.

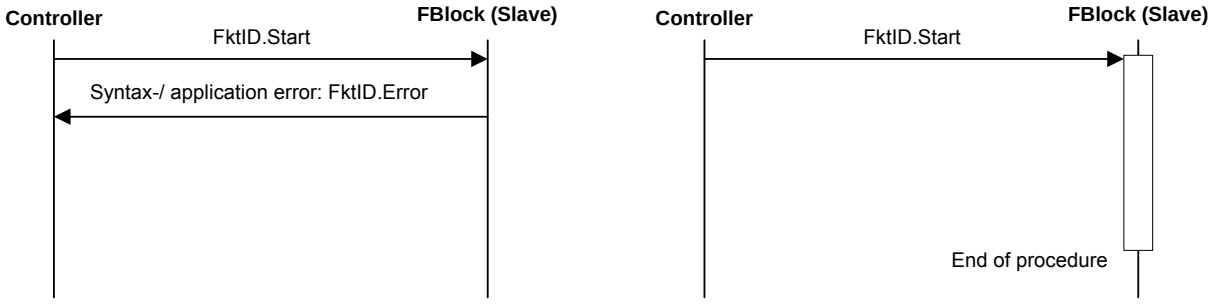

Figure 2-19: Sequences when using Start with and without error

A method started by “Start” must be called only one time (multiple instances are not allowed). In case a method that was started by “Start” is currently running, and a second Controller tries to start the same method again, the method has to reply with an error “Busy”. The already running method is not affected by this new incoming request. For running several instances of the same method, StartAck must be used.

Note: *Usage of non-Ack OPTypes is no longer recommended. In the above example, the OPType StartAck should be used instead of Start.*

2.2.3.5.4 StartResult, Result, Processing, Error

In opposite to triggering a method by using Start, the Controller requires feedback when it uses StartResult. It then expects reports about the currently running procedure (with OPType Processing), as well as about the result (with OPTypes Result or Error).

If there are syntax or parameter errors during the calling of a method, there will be a reply using Error. The method will not be started.

If a method that was started can generate a result within $t_{ProcessingDefault1}$ after reception of StartResult, it returns the result by using “Result(<Parameter>)” as soon as it is available. There will be no reply “Processing” in that case. The same applies to application errors.

If a method cannot generate a result within $t_{ProcessingDefault1}$ after having received StartResult and if there is no application error, it replies after that time by using “Processing”. After that, it starts the timer $t_{ProcessingDefault2}$. This timer works in the same way as $t_{ProcessingDefault1}$. That means that in case of terminating the method within $t_{ProcessingDefault2}$, a reply “Result(<Parameter>)” will be sent. Otherwise, “Processing” will be reported when the timer expires. Upon sending processing, the timer is restarted.
The Controller evaluates the first reply by using a timer interval of $t_{WaitForProcessing1}$ (compensation of eventual delays). If there is no reply within this time (neither Result, nor Error, nor Processing), it assumes an error. After receiving the first processing, it uses a timer with interval $t_{WaitForProcessing2}$ for the following receptions.

The System Integrator may change the two default timing values for each method:

1. Initial timeout between StartResult and Processing ($t_{ProcessingDefault1}$)
2. A second timeout between subsequent processing messages ($t_{ProcessingDefault2}$)

Note: *Usage of the Start and StartResult OPTypes is no longer recommended for Application FBlocks. The OPTypes StartAck and StartResultAck should be used instead.*

2.2.3.5.5 Get, Status, Error

By using OPType Get, a Controller asks for the status of a property. In case of a Get request, a reply using Status will be generated, if the syntax check has shown no errors. Otherwise, Error will be returned. A property shall reply to a request within $t_{Property}$. If the Controller does not receive a reply within $t_{WaitForProperty}$ after having sent Get, this is an error. The Controller may react more tolerantly and wait for a longer time. Nevertheless, an interruption of the waiting process is required.

2.2.3.5.6 Set, Status, Error

By using Set, the content of a property is changed. Set behaves equal to StartAck for methods. This means that the Controller does not expect any reply (except error reports). If the syntax check is ok, the command can be executed.

The changed status of the property will be reported to all Controllers that are registered for this function. This is done via notification. If the triggering Controller is registered, it will receive a status report indirectly. This way is recommended, for example, if the Controller is registered in the Notification Matrix. In addition, it may be that the changing of a property, by a Controller from outside, generates the changing of the status of several other properties by some internal mechanisms.

Therefore, the controlling of properties by using Set is the preferred mechanisms for Controllers that are registered in the Notification Matrix of a controlled FBlock.

2.2.3.5.7 SetGet, Status, Error

SetGet is the preferred way of controlling FBlocks for Controllers:

- that control a property only in rare cases
- that are not registered in the Notification Matrix

SetGet is a combination of Set and Get, which means that the Controller (in case of a correct syntax) automatically gets the changed status in return.
In case of a request by using SetGet, a reply using Status is generated, if the syntax check has shown no errors. Otherwise Error will be returned. A property shall reply on a request within $t_{Property}$. If the Controller does not receive any reply within $t_{WaitForProperty}$ after having sent SetGet, an error can be assumed. It is not critical if the Controller reacts more tolerantly and waits for a longer time. Nevertheless, an interruption of the waiting process is required.

2.2.3.5.8 GetInterface, Interface, Error

With the OPType GetInterface the Function Interface is requested, which is returned with the OPType Interface. The behavior of these OPTypes is similar to Get and Status (refer to section 2.2.3.5.5).

Note: *GetInterface and Interface are optional OPTypes (see 4 Appendix A: Optional OPTypes).*

2.2.3.5.9 Increment and Decrement, Status, Error

Increment and Decrement provide a relative changing of a variable in opposite to the absolute changing by using Set. When using Increment or Decrement, the new status will be reported to the triggering Controller, as well as to the Controllers registered in the Notification Matrix. This is similar to SetGet. In case of a Controller requesting Increment or Decrement although the respective maximum or minimum is reached, no error will be reported. In fact the old value will be reported. This answer is directed to the triggering Controller only. A reporting to the Controllers registered in the Notification Matrix is not required since the value actually did not change.

Note: *Usage of the OPTypes Increment and Decrement is not recommended in conjunction with multicast messages; the OPType SetGet should be used instead.*

2.2.3.5.10 Abort, Error

These OPTypes are available for methods only. When used, Abort terminates the execution of a method. The message abortion is confirmed through an Error(Aborted) message. Abort must not have any parameters. Please note that methods in general should be aborted only by that application which has started the method. After the method has been aborted, information about this is sent out. Please see *8 Application Error – Method Aborted (ErrorCode 0x43)* on page 61 for more information.

Note: *Usage of the Abort OPType is no longer recommended. The OPType AbortAck should be used instead.*

2.2.3.5.11 AbortAck, ErrorAck

This OPType is available for methods only. When used, AbortAck terminates the execution of a method. The message abortion is confirmed through an ErrorAck(Aborted) message. In opposite to “Abort”, AbortAck transports additional “routing” information (SenderHandle, as described in section 2.2.2.2 on page 48). AbortAck must not have any parameters except SenderHandle.
Please note that methods in general should be aborted only by that application which has started the method. After the method has been aborted, information about this is sent out. Please see 8 Application Error – Method Aborted (ErrorCode 0x43) on page 61 for more information.

2.2.3.6 Function Formats in Documentation

In documentations that must be human readable, messages have to be presented in a suitable form:

```
SrcAdr -> TrgAdr: FBlockID.InstID.FktID.OPType(Parameter)
```

SrcAdr and TrgAdr are the node position addresses of the sending and the receiving device, respectively. In most cases, only one instance of the FBlock is available in the system and InstID can be omitted.

```
SrcAdr -> TrgAdr : FBlockID.FktID.OPType(Parameter)
```

Example:

Choosing track of the CD changer:

```
HMI -> CDC : AudioDiskPlayer.Track.Set(5)
CDC -> HMI : AudioDiskPlayer.Track.Status(5)
```

The messages are included in a library and are grouped by functions (FBlock Library).

2.2.3.7 Length

Length specifies the length of the data field in bytes. It is encoded in 16 bits.

Length = 0x0000 Data field of length 0
Length = 0x0001 Data field of length 1 byte.
Length = 0xFFFF Data field of length 65535 bytes.

The system-specific maximum length is defined by the System Integrator. Functions that need to transport large application messages communicate via MOST High Protocol and the Packet Data Transfer Service. These functions will be marked in the FBlock Specification.

For the description of length determination, please refer to 3.2.5.2 Application Message Service (AMS).

2.2.3.8 Data and Basic Data Types

Within a data field, none, one, or multiple parameters in any allowed combination of the following data types can be transported. They are transported most significant bit (MSB) first. The sign is encoded in the most significant bit and 2's complement coding is used for signed values. There are the following basic data types:

Boolean	Unsigned Byte	String
BitField	Signed Byte	Stream
Enum	Unsigned Word	Classified Stream
	Signed Word	Short Stream
	Unsigned Long	
	Signed Long	

Floating point representation
Using only the data types mentioned above, no floating point format would be possible. The missing information about the location of the decimal point is added via an exponent of type Signed Byte.

Note: *The exponent is a parameter-specific constant. It is therefore not transmitted over MOST when the value is transmitted.*

The value to be displayed must be transported in the following way:

value to be displayed = transmitted value * $10^{Exponent}$

Example 1:

transmitted value:	1073 (word)
exponent:	-1
step:	1
unit:	MHz
value to be displayed:	107.3 (MHz) (can be changed in steps of 100 kHz)

In case of an Increment operation with NSteps = 5, the current frequency would be incremented from 107.3 MHz to 107.8 MHz.

Example 2:

transmitted value:	1073 (word)
exponent:	+5
step:	1
unit:	Hz
value to be displayed:	107,300,000 (Hz) (can be changed in steps of 100 000 Hz)

In case of an Increment operation with NSteps = 5, the current frequency would be incremented from 107,300,000 Hz to 107,800,000 Hz.

Example 3:

transmitted value:	1000 (word)
exponent:	-3
step:	10
unit:	m
value to be displayed:	1.000 (m) (can be changed in steps of 10 mm)

In case of an Increment operation with NSteps = 5, the current length would be incremented from 1.000 m to 1.050 m.

The exponent can be already known through the receiver of the parameter (Controller), or it can be requested through the sender (function) of the value (refer to section 2.2.4 on page 77).

2.2.3.8.1 Boolean

Definition of Type	Comments
1 byte	Only one bit of the byte can be used.

2.2.3.8.2 BitField

Definition of Type	Comments
Size byte	= (Mask.Data)

Size: - (Total Size of the BitField): 1, 2, 4, or 8 bytes

Data: ½ Size byte (Data Content Area)

Mask: ½ Size byte (Masking Area):
"Mask" is a masking bit field of the same size as the Data Content Area "Data". It indicates to which bits in the Data Content Area of the BitField an operation shall be applied. The LSB of "Mask" masks the LSB of the Data Content:

bit k (Mask) = 1 -> apply Operation to bit k (Data)
bit k (Mask) = 0 -> do not apply operation to bit k (Data)

Example:

```
State:      MyBitField.Status (XXXX XXXX, 1010 1001)
Operation:  MyBitField.Set    (0000 1000, 1010 0111)
NewState:   MyBitField.Status (XXXX XXXX, 1010 0001)
```

"X" means "don't care" in this example. These bits should be set to zero by the sender of the Status message. However, their content must be ignored in the receiver of the Status message.

BitField in Arrays and Records:

A BitField in an Array or Record represents one variable. That means it is addressable as an entity via one dedicated value of Pos.

Example:

```
MyArray = Array of BitField:  XXXX XXXX,0100 1001
                              XXXX XXXX,1110 0011
                              XXXX XXXX,0010 1101
                              XXXX XXXX,0111 1111
```

Requesting Status report (1):

```
MyArray.Get (PosX=0x0)
```

Answer:

```
MyArray.Status (PosX=0x0,     XXXX XXXX,0100 1001,
                              XXXX XXXX,1110 0011,
                              XXXX XXXX,0010 1101,
                              XXXX XXXX,0111 1111)
```

Requesting Status report (2):

```
MyArray.Get (PosX=0x2)
```

Answer:

```
MyArray.Status (PosX=0x2,     XXXX XXXX,1110 0011)
```

Performing a Set operation (1):

```
MyArray.Set (PosX=0x0,        1000 0001,1000 0001,
                              1000 0001,1000 0001,
                              1000 0001,0111 1110,
                              1000 0001,0111 1110)
```

Result:

```
MyArray =                     XXXX XXXX,1100 1001
                              XXXX XXXX,1110 0011
                              XXXX XXXX,0010 1100
                              XXXX XXXX,0111 1110
```

Performing a Set operation (1):

```
MyArray.Set (PosX=0x4,        1111 0000,1000 0001)
```

Result:

```
My Array =                    XXXX XXXX,1100 1001
                              XXXX XXXX,1110 0011
                              XXXX XXXX,0010 1100
                              XXXX XXXX,1000 1110
```

2.2.3.8.3 Enum

Definition of Type	Comments
1 byte	-

2.2.3.8.4 Unsigned Byte

Definition of Type	Comments
1 byte	-

2.2.3.8.5 Signed Byte

Definition of Type	Comments
1 byte	-

2.2.3.8.6 Unsigned Word

Definition of Type	Comments
2 bytes	-

2.2.3.8.7 Signed Word

Definition of Type	Comments
2 bytes	-

2.2.3.8.8 Unsigned Long

Definition of Type	Comments
4 bytes	-

2.2.3.8.9 Signed Long

Definition of Type	Comments
4 bytes	-

2.2.3.8.10 String

Definition of Type	Comments
Variable length	= (Identifier.Content.Terminator)

In general, only “most significant bit (MSB) first, high byte first” notation must be used for strings. Every string starts with an Identifier and is Null terminated.

Identifier: 1 byte

Code	String type	ASCII compatible
0x00	Unicode, UTF16	No
0x01	ISO 8859/15 8bit	Yes
0x02	Unicode, UTF8	Yes
0x03	RDS	No
0x04	DAB Charset 0001	No
0x05	DAB Charset 0010	No
0x06	DAB Charset 0011	Yes
0x07	SHIFT_JIS	No
0x08 - 0xBF	Reserved	
0xC0…0xEF	System Integrator (e.g., Car Maker)	
0xF0…0xFF	Supplier	

Content: Characters

Terminator: 1 Character Null character. Number of zeros. Depends on encoding.

For calculating length, only the number of characters is relevant. Length explicitly excludes the Identifier and the terminating character(s). Strings that are using the RDS character set may contain codes for switching the code pages. This can produce strings, which need more bytes in memory than the number of characters they contain.
The encoding of an “empty” string depends on the used code:

Code	“Empty” String	Comment
UNICODE, UTF16	0x00,0x00,0x00	-
ISO 8859/15 8 bit	0x01,0x00	-
Unicode, UTF8	0x02,0x00	-
RDS	0x03,0x00	-
SHIFT_JIS	0x07,0x00,0x00	-

Since all strings are null terminated, character sets that use a null character are not allowed.

2.2.3.8.11 Stream

Definition of Type	Comments
Variable length	Any data content

Streams can be unstructured or structured. An unstructured stream contains arbitrary data. A structured stream is closely related to the notion of a Variant Record (disjoint union) in common programming languages. In MOST, these are called “stream cases”.

Unstructured streams
Unstructured streams do not contain any stream cases.

Structured streams
A structured stream consists of one or multiple stream cases. The individual stream cases contain either a series of stream parameters or stream signals.

Simple Stream
A simple stream is either an unstructured stream or a structured stream with exactly one stream case.

Complex Stream
A complex stream is a structured stream that consists of multiple stream cases.

The following restrictions apply to the use of data type Stream:

1. If a parameter is of data type Stream, this parameter must be the last parameter of a message.
2. If a member of a record is of data type Stream, this member must be the last element of the record.
3. Data type Stream cannot be used in any Array function class (including DynamicArrays, LongArrays and Maps).

2.2.3.8.11.1 Stream Cases

A selector value determines which defined type is used, that is, which stream case applies.

- The selector value itself is not part of the Stream and must be in front of the Stream it controls; however, the selector of a complex stream may be embedded in another stream.
- Other parameters may occur between the selector and the Stream.
- A string selector must be coded with string code 0x01 (ISO8859/15 8bit).

Example:
In this example, information about a contact is to be transmitted over MOST. The contact could be a real person, a company, or an institution. Depending on the kind of contact (ContactType), the number of transmitted data items (ContactData) will vary.

ContactType: The ContactType Enum is the selector. If, for instance, the value of parameter ContactType is 0x00, the contact is a person.

Basis datatype	Range of values	Code	Description
Enum	0x00…0x02	0x00	Person
		0x01	Company
		0x02	Institution

ContactData: This is the actual contact information. Depending on the value of the selector (ContactType), ContactData will contain different items. If, for example, ContactType is 0x00 (the contact is a person), ContactData will contain LastName, MiddleName, and FirstName.

Basis datatype	Length	Condition	Description
Stream		ContactType = 0x00	Content: LastName, MiddleName, FirstName
		ContactType = 0x01	Content: CompanyName, CompanyType
		ContactType = 0x02	Content: InstitutionName

2.2.3.8.11.2 Stream Signals

A stream signal represents a value that consists of an arbitrary number of bits.

Example:
In this example, the state of one 4-bit port and two 2-bit ports is to be transmitted over MOST in a Stream.

PortState: The PortState stream is divided into three stream signals. One of them is 4 bits wide, two of them are 2 bits wide.

Basis datatype	Max. Length	Condition	Description
Stream		-	Stream signals: Port0, Port1, Port2

Signal Name	Bit position	Number of bits	Description
Port0	0	4	State of 4-bit Port .
Port1	4	2	State of 2-bit Port 1.
Port2	6	2	State of 2-bit Port 2.

2.2.3.8.12 Classified Stream

Definition of Type	Comments
Variable length	=(Length.MediaType.Content)

Classified Stream acts as a container for different objects.

Length: 2 bytes Length of MediaType and Content in bytes.

MediaType: Null terminated ASCII string (no coding identifier) containing the data typing of the object that is transported in the Classified Stream. The format used for this is the same as for HTTP/1.1.
MediaType = type "/" subtype *(";" parameter)

The MediaType's values type, subtype, and parameter are specified by the Internet Assigned Number Authority IANA.

Example:

	Length		MediaType											Content			
Byte Nr.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Data	0x00	0x0F	t	e	x	t	/	p	l	a	i	n	0x00	M	O	S	T

When MediaType is an empty string, "application/octet-stream" shall be assumed:

	Length		MediaType	Content											
Byte Nr.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Data	0x00	0x0D	0x00	< *binary data* >											

Information about HTTP/1.1 can be found in:
RFC 2616 - Hypertext Transfer Protocol — HTTP/1.1, R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. June 1999. (Obsoletes RFC 2068).

2.2.3.8.13 Short Stream

Definition of Type	Comments
Variable length	=(Length.Content)

The Short Stream type supports the same structuring mechanisms as the Stream data type, that is, stream cases and stream signals.

Length: 1 byte Length of the Content.

Content: max. 255 bytes Any data.

2.2.4 Function Classes

When having a look at function classes, properties and methods must be differentiated. The properties themselves consist of such with one variable, and of such with multiple variables.

In the next sections, the following universal parameters are used by the optional OPTypes GetInterface and Interface:

Flags: 8 bits

Bit 6-7	Bit 4-5	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Channel Type	Notification	Unicode	Enabled	Visible

By using the Visible bit, the device can influence whether the function is displayed at the moment or not (default = 1 = visible). It is possible to disable a function temporarily. This is done by setting the Enabled bit to 0. Bit 2 indicates whether a function uses Unicode or standard strings. The Notification bit shows whether a function supports notification. This bit is valid only for properties. The Channel Type bit field consists of two bits. It shows the type of channel that is used when communicating with the function. Table 2-7 shows the three possible modes.

OPType	Property	Method	Mode 0	Mode 1	Mode 2
0	Set	Start	C	A	A/C
1	Get		C	A	C
		Abort	C	C	C
2	SetGet	StartResult	C	A	A/C
3	Increment		C	C	C
4	Decrement		C	C	C
5	GetInterface[1]	GetInterface[1]	C	C	C
6		StartResultAck	C	A	A/C
7		AbortAck	C	C	C
8		StartAck	C	A	A/C
9		ErrorAck	C	C	C
A		ProcessingAck	C	C	C
B		Processing	C	C	C
C	Status	Result	C	A	A/C
D		ResultAck	C	A	A/C
E	Interface[1]	Interface[1]	C	C	C
F	Error	Error	C	C	C

Table 2-7: The different modes of the bit field Channel Type

The OPTypes GetInterface and Interface need to be available on the Control Channel so that the interface can be received regardless of if the requesting node is using MOST High or not. Furthermore, MOST High Protocol connections must not be established for the OPTypes Increment, Decrement, AbortAck, ProcessingAck, Processing, Error, and ErrorAck.
MOST High Protocol connection errors and syntax errors (see section 2.2.3.5.1) are implicitly indicated by the failure to open a MOST High connection. If syntax errors and application errors are reported, this must be done via the Control Channel.

The meaning of the characters “C” and “A” in the table is as follows:
C: messages on Control Channel
A: messages on the Packet Data Channel using MOST High
A/C: messages either on the Control Channel or the Packet Data Channel using MOST High

[1] GetInterface and Interface are optional OPTypes.

Mode 0:
This is the standard mode where all communication with the function is done via the Control Channel.

Mode 1:
Main communication is done via the MOST High Protocol on the Packet Data Channel.

Mode 2:
This is a mixed mode where only the OPTypes that are carrying a lot of data are accessed over the Packet Data Channel via the MOST High Protocol.

Bits 6 and 7 are reserved for future use.

The following table describes the remaining parameters.

Parameter	Size	Code	Description
Class	8 bits	0x00	Unclassified Method
		0x01	Trigger Method
		0x02	Sequence Method
		0x10	Unclassified Property
		0x11	Switch
		0x12	Number
		0x13	Text
		0x14	Enumeration
		0x15	Array (Refer to section 2.2.4.2.2 on page 89.)
		0x16	Record (Refer to section 2.2.4.2.1 on page 87.)
		0x17	DynamicArray (Refer to section 2.2.4.2.3 on page 92.)
		0x18	LongArray (Refer to section 2.2.4.2.4 on page 95.)
		0x19	BoolField
		0x1A	BitSet
		0x1B	Container
		0x1C	Sequence Property
		0x1D	Map
		0xFF	Abort (No further specifications behind this location)
OPTypes	16 bits		Flag field of available OPTypes (1 = OPType available). LSB represents the least significant OPType “Set”, which has code 0x0.
Name			Name of function as null terminated string.

Table 2-8: Parameters for Interface descriptions: Class, OPTypes, and Name

2.2.4.1 Properties with a Single Parameter

Many functions contain only a single parameter. These functions can be divided into classes, which correspond with the type declaration in programming languages. The class of a property is derived from the basis data type (Refer to section 2.2.3.8 on page 69) of its variable.

At the moment there are the following function classes for single properties:

Function Class	Explanation
Switch	Properties of this class contain a variable of type Boolean (on/off; up/down). It can be set (Set, SetGet) or read (Get, Status).
Number	Properties of this class contain a numeric variable (frequency, speed limit, temperature), which can be read (Get, Status), set absolutely (Set, SetGet) or changed relatively (Increment, Decrement).
Text	Properties of this class have a string variable (Status), e.g., Warning, Hint.
Enumeration	Properties of this class contain a variable of type Enum. They provide an unchangeable number of invariable elements, from which can be chosen (Set). Examples: Drive status (Stop, Pause, Play, Forward, Rewind), Dolby (B, C, Off).
BoolField	Properties of this class contain a number of bits that should either be used as flag field, or as controlling bits that are always manipulated together.
BitSet	Properties of this class are based on data type BitField. They contain a number of bits, which can be manipulated individually.
Container	Properties of this class contain either a Stream, Classified Stream, or Short Stream.

Table 2-9: Classes of functions with a single parameter

The function classes (basic classes) with one variable and their resulting protocols are described in detail below.

2.2.4.1.1 Function Class Switch

OPType	Parameters
Set	**Content**[1]
Get	
SetGet	Content
GetInterface[2]	
Status	Content
Interface	Flags, Class, OPTypes, Name
Error	ErrorCode, ErrorInfo

Content:	Boolean	0 for off 1 for on

Example: RDSOnOff in AM/FMTuner1

Function:	**RDSOnOff**	**e.g., 0x00A**
Flags:	**visible, enabled, no Unicode, notification**	**0000 1011 = 0x0B**
Class:	**Switch**	**0x11**
OPTypes:	**Get, SetGet, Status GetInterface, Interface, Error**	**1101 0000 0010 0110 = 0xD026**
Name:	**RDSOnOff**	**"RDS"**

Setting RDS = OFF:

```
HMI -> Tuner: AM/FMTuner.1.RDSOnOff.SetGet (00)
```

Note: *This is a hypothetical example. It does not necessarily follow the MOST FBlock Specification.*

[1] In the next sections that introduce individual function classes, some parameters appear in **boldface** type inside the OPType tables. These marked parameters are characteristic for a function class and are described in detail below the tables.
[2] Interface and GetInterface are optional OPTypes. This applies to function class Switch and all function classes described in the following sections.

2.2.4.1.2 Function Class Number

OPType	Parameters
Set	**Content**
Get	
SetGet	Content
Increment	**NSteps**
Decrement	NSteps
GetInterface	
Status	Content
Interface	Flags, Class, OPTypes, Name, **Units, DataType, Exponent, Min, Max, Step**
Error	ErrorCode, ErrorInfo

Content:	Unsigned Byte Signed Byte Unsigned Word Signed Word Unsigned Long Signed Long	The value of this property.
DataType:	Unsigned Byte	Type of variable: 0x00 Unsigned Byte 0x01 Signed Byte 0x02 Unsigned Word 0x03 Signed Word 0x04 Unsigned Long 0x05 Signed Long
Exponent:	Signed Byte	Position of decimal point; Value = Number * $10^{Exponent}$
Min:		Minimum value of variable of type *DataType*
Max:		Maximum value of variable of type *DataType*
Step:		Step width for adjusting type *DataType.* The following condition must always be true: Max = Min + (n * Step)
NSteps:		Unsigned Byte Number of steps, as defined under “Step width for adjusting”. Default value is 1; value 0 is not allowed. NSteps has no exponent, but has the same unit as the Number parameter.
Units:	Unsigned Byte	Unit, according to Table 2-10.

Unit	Encoding
none	0x00
Distance:	
cm	0x01
m	0x02
km	0x03
miles	0x04
Time:	
µs (Micro second)	0x10
ms (Millisecond)	0x11
s (Second)	0x12
min (Minute)	0x13
h (Hour)	0x14
d (day)	0x15
mon (Month)	0x16
a (Year)	0x17
Frequency:	
1/min	0x20
Hz	0x21
kHz	0x22
MHz	0x23
Volume:	
l (Liter)	0x30
gal (UK)	0x31
gal (US)	0x32
ccm	0x33
Consumption:	
l/100km	0x40
miles/gal	0x41
km/l	0x42
Speed and Acceleration:	
km/h	0x50
miles/h	0x51
m/s	0x52
cm/s	0x53
°/s	0x54
m/s^2	0x55

Unit	Encoding
Temperature and Pressure:	
°C	0x60
F	0x61
K	0x62
bar	0x63
psi	0x64
Miscellaneous:	
dB	0x70
%	0x71
Voltage:	
mV	0x80
V	0x81
Current:	
mA	0x90
A	0x91
Angle:	
Degrees	0xA0
Minutes	0xA1
Seconds	0xA2
$360°/ 2^{32}$	0xA3
$360°/ 2^{8}$	0xA4
Resolution:	
Pixel	0xB0
Data:	
Byte	0xC0
kByte	0xC1
MByte	0xC2
GByte	0xC3
TByte	0xC4
bit	0xC5
Data Rate:	
bps	0xD0
kbps	0xD1
Mbps	0xD2
Bps	0xD3
kBps	0xD4
MBps	0xD5

Table 2-10: Available units[1]

[1] According to the International System of Units (SI).

2.2.4.1.3 Function Class Text

OPType	Parameters
Set	**Content**
Get	
SetGet	Content
GetInterface	
Status	Content
Interface	Flags, Class, OPTypes, Name, **MaxSize**
Error	ErrorCode, ErrorInfo

Content:	String	The actual text.
MaxSize:	Unsigned Byte	Maximum length of the string

2.2.4.1.4 Function Class Enumeration

OPType	Parameters
Set	**Content**
Get	
SetGet	Content
GetInterface	
Status	Content
Interface	Flags, Class, OPTypes, Name, **Size, Name1, Name2...**
Error	ErrorCode, ErrorInfo

Content:	Enum	Number of active element or of element to be activated
Size:	Unsigned Byte	Length of enumeration 0 = no element 1 = one element 2 = two elements....
Name x:	String	Null terminated string, representing the name of element x.

2.2.4.1.5 Function Class BoolField

OPType	Parameters
Set	**Content**
Get	
SetGet	Content
GetInterface	
Status	Content
Interface	Flags, Class, OPTypes, Name, **DataType, NElements, BitName, BitSize, BitName, BitSize, ...**
Error	ErrorCode, ErrorInfo

Content:	Unsigned Byte Unsigned Word Unsigned Long	Data area, containing, e.g., flags
DataType:	Unsigned Byte	Type of variable: 0x00 Unsigned Byte 0x02 Unsigned Word 0x04 Unsigned Long
NElements:	Unsigned Byte	Number of Elements in the BoolField
BitName:	String	Null terminated string, indicating the name of the respective element
BitSize:	Unsigned Byte	Number of bits required for encoding the element. Encoding starts at the LSB.

When function class BoolField is used, a field of either 8bits, 16bits, or 32bits will be reserved. Using the flags starts at the least significant bit (LSB). The value 0b**** ***0 means false and 0b**** ***1 means true. Manipulating a BoolField always requires the writing of the entire field.

Example:
This example shows a BoolField based on “Unsigned Word”. There are 11 bits used for representing some flags. Please note that it is also possible to combine several bits for representing a special element (flag).

					B	Y	T	E	1
D 7	D 6	D 5	D 4	D 3	D 2	D 1	D 0		
					F. 10	F. 9	F. 8		

					B	Y	T	E	0
D 7	D 6	D 5	D 4	D 3	D 2	D 1	D 0		
F. 7	F. 6	F. 5	F. 4	F. 3	F. 2	F. 1	F. 0		

2.2.4.1.6 Function Class BitSet

OPType	Parameters
Set	**Content**
Get	
SetGet	Content
GetInterface	
Status	Content
Interface	Flags, Class, OPTypes, Name, **Size**
Error	ErrorCode, ErrorInfo

Content:	BitField	
Size:	Unsigned Byte	Size of SetOfBits (Mask + Data) in bytes

2.2.4.1.7 Function Class Container

OPType	Parameters
Set	**Content**
Get	
SetGet	Content
GetInterface	
Status	Content
Interface	Flags, Class, OPTypes, Name, **Datatype**, **MaxLength**
Error	ErrorCode, ErrorInfo

Function class Container is used for objects that cannot be described in a satisfying way by the other structures.

Content:	Stream Classified Stream Short Stream	
DataType:	Unsigned Byte	Type of variable: 0x00 Stream 0x01 Classified Stream 0x02 Short Stream
MaxLength:	Unsigned Word	MaxLength indicates the maximum size of the stream in bytes.

2.2.4.2 Properties with Multiple Parameters

Some functions contain multiple parameters. Here the principle should be to only combine functions that are very similar in nature (e.g., Station name and PI). Parameters that do not match like that should be modeled in separate functions (e.g., Station name and current frequency).

Functions with multiple parameters can also be assigned to classes called Array and Record. In an Array, parameters are of the same type, in a record they are of different types. It is possible to build an Array of Records or a Record containing an Array. Such “two dimensional” constructs are allowed.

More complex constructs with dimensions exceeding two (Array of Array of Record or a Record with two Arrays) are definitely not allowed. In addition, it is not allowed to reference other functions from within a function. This means that an interface description of a function must not reference the interface descriptions of other functions. A function must be described completely and independent of other functions.

Function Class	Explanation
Record	Properties of this class contain a variable of a composite type. It may consist of any number of single properties.
Array	Properties of this class contain only elements of the same type.
DynamicArray	Properties of this class use a more dynamic approach than ordinary Arrays.
LongArray	Properties of this class are used to handle large Arrays in a sophisticated way.
Sequence Property	Properties of this class consist of a number of single values that can be treated together as a single property.
Map	Properties of this class are similar to DynamicArrays; however, no ordering of the elements may be assumed.
Unclassified Property	Unclassified Properties cannot be matched to any other function class in a meaningful manner.

Table 2-11: Classes of functions with multiple parameters.

Note:

1. *All basic data types, except the data type ‘Stream’ can be used without restrictions as parameters in Arrays, Records and Sequences. Stream does not have a defined length and is therefore excluded for Arrays. Stream may be used as last element in Records or as last parameter in a Sequence.*
2. *In the interface description of function classes with multiple parameters, the OPTypes are omitted.*

2.2.4.2.1 Function Class Record

OPType	Parameters
Set	**Position**, **Data**
Get	Position
SetGet	Position, Data
Increment	Position, NSteps
Decrement	Position, NSteps
GetInterface	
Status	Position, Data
Interface	Flags, Class, Name, **NElements, IntDesc1, IntDesc2...**
Error	ErrorCode, ErrorInfo

Position always consists of two bytes and indicates what will be set, requested, or read in the record. The first byte (x) indicates the position of an element in the record. If the record contains an Array (two dimensions), the second byte specifies the element in the Array. On:

- x=y=0,
  the operation is related to the entire record.
- x=(Position of Array in Record) AND y>0,
  the operation is related to the y-th element in the Array.
- x=(Position of Array in Record) AND y=0,
  the operation is related to the entire Array.
- x<>(Position of Array in Record) AND y=0,
  the operation is related to the respective element in the record.

Even if the record does not contain an Array, the position consists of two bytes, but the second byte is not used in this case.

Data represents data according to the structure of the record and the specifications by position.

If a record contains an Array, the Array must be the last element of the record.

NElements	Unsigned Byte	Number of elements in Record

IntDescX are the interface descriptions of the single elements. Depending on the data type, one of the interface descriptions defined for the respective class can be inserted. Please note that here in case of elements, parameter Flags is not available. For parameter OPTypes only Set, Get, SetGet, Status, Increment, Decrement, and Error can be used.

Note: *IntDesc only represents a group of parameters. No referencing of other functions and their interface descriptions is done here!*

Below, IntDesc is displayed with respect to the basic classes:

Class	IntDesc
Switch	Class, OPTypes, Name
Number	Class, OPTypes, Name, **Units, DataType, Exponent, Min, Max, Step**
Text	Class, OPTypes, Name, **MaxSize**
Enumeration	Class, OPTypes, Name, **Size, Name1, Name2, ...**
BoolField	Class, OPTypes, Name, DataType, **NElements, BitName, BitSize, BitName, BitSize, ...**

Class	IntDesc
BitSet	Class, OPTypes, Name, **Size**
Array	Class, Name, **NElements, IntDesc**
Container	Flags, Class, OPTypes, Name, **MaxLength**

Even if the record does not contain an Array, the position consists of two bytes, but the second byte is not used in this case.

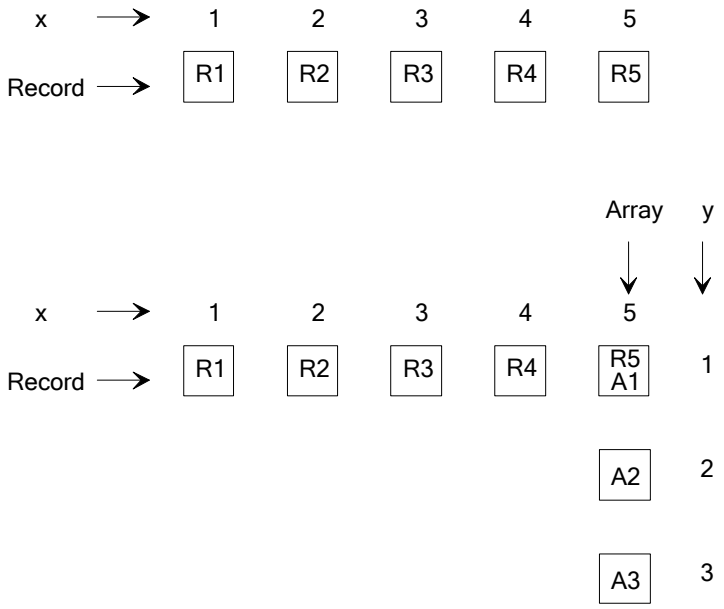

Figure 2-20: Meaning of position x in a record (above) and of position y in a record with an Array (below)

2.2.4.2.2 Function Class Array

OPType	Parameters
Set	Position, Data
Get	Position
SetGet	Position, Data
Increment	Position, NSteps
Decrement	Position, NSteps
GetInterface	
Status	Position, Data
Interface	Flags, Class, Name, **NMax, IntDesc**
Error	ErrorCode, ErrorInfo

Function class Array is very similar to Record. **NMax**, of type Unsigned Byte, represents the maximum number of elements. Since the Array contains only elements of the same type, there only needs to be one IntDesc of the following type:

Class	IntDesc
Switch	Class, OPTypes, Name
Number	Class, OPTypes, Name, **Units, DataType, Exponent, Min, Max, Step**
Text	Class, OPTypes, Name, **MaxSize**
Enumeration	Class, OPTypes, Name, **Size, Name1, Name2, ...**
BoolField	Class, OPTypes, Name, DataType, **NElements, BitName, BitSize, BitName, BitSize, ...**
BitSet	Class, OPTypes, Name, **Size**
Array	Class, Name, **NElements, IntDesc**
Record	Class, Name, **NElements, IntDesc1, IntDesc2, ...**
Container	Flags, Class, OPTypes, Name, **MaxLength**

Analogous to the determinations of a record, the following is valid here for an Array:

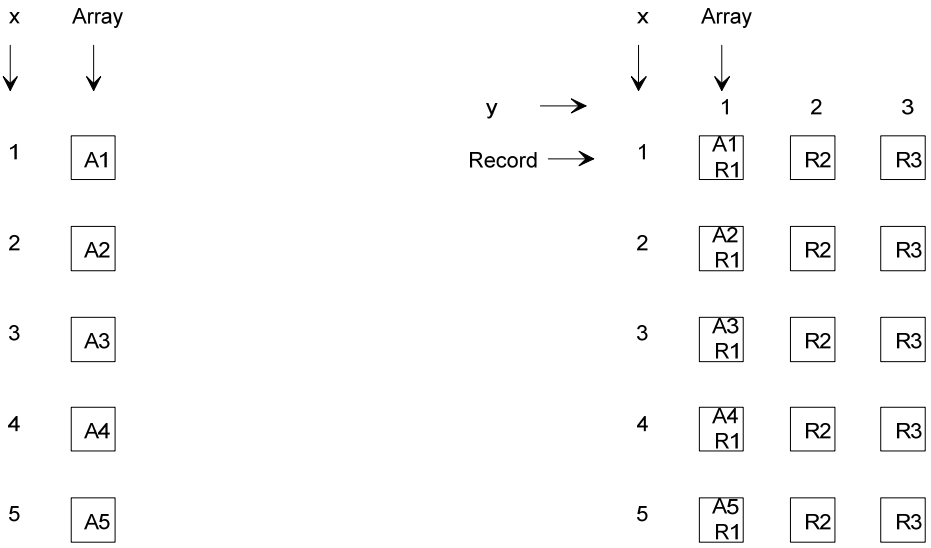

Figure 2-21: Position x in case of an Array of basic type (left), y in case of an Array of Record (right)

As in the case of a record, Position always consists of two bytes, independent of whether the Array contains a record or not. If there is no record, the second byte is not used.

The first parameter x (first byte) always refers to the outer structure, that is, the Array for an Array of Record, and the record for a Record with Array.

If a partial structure is transmitted by using Position, the sending device is responsible for keeping consistency with the general structure transmitted before. As an example, the AM/FMTuner may update the signal qualities in a station list that was transferred earlier. It must make sure that the signal quality values are assigned to the correct stations.

Transmitting an Array is the only time when it is possible to transmit fewer elements than the maximum number of elements. As an example, on 10 receivable stations the entire list of perhaps 100 possible entries does not need to be transferred. It must be kept in mind that each individual element of the Array must always be transferred completely. If not, this is an error. The specification of the length is done in parameter Length of the application protocol.

If an Array is empty, the status is reported without data:

```
FBlockID.InstID.Array.Status (PosX=0x00, PosY=0x00)
```

Examples:

Disk information in CD changer:

The CD changer contains a magazine of up to 10 CDs. Each disk contains several tracks. The information is modeled in the two properties Magazine and Disk.

Magazine = Array[1..10] of Record of

DiskTitle:	String	(Text)
TotalTime:	Int	(Number)
NTracks:	Unsigned Byte	(Number)

If a disk is not available, this can be recognized by TotalTime and NTracks containing 0x00.

Disk = Array[1..99] of Record of

TrackTitle:	String	(Text)
TrackTime:	Unsigned Byte	(Number)

On request

```
Controller -> CDC: AudioDiskPlayer.0x01.Magazine.Get (03. 01)
```

one receives the title of the third disk. On request

```
Controller -> CDC: AudioDiskPlayer.0x01.Magazine.Get (00. 01)
```

the titles of all disks are returned.

Selecting In Arrays:

In many Arrays, lines will be selected. Here, selections “1 of n” (one single line selected only) need to be differentiated from selections “n of N” (several lines can be selected at the same time).

- n of N:
  The selection here should be done by an individual parameter Selected of type Switch, which is used as prefix (Array of Record of {Selected, ...}). The change in the status of the switch can be modified by Controller or Slave either single (Selected of a single line), or for an entire column (Selected of all lines). In principle, this kind of selection can be used in case of 1 of N as well.

- 1 of N:
  In case of 1 of N there is an alternative modeling which is less expensive with respect to communication than n of N. Here a property Selected is modeled, which points onto the selected line. The kind of pointer differs individually. So, for example, in case of station lists the pointer may point onto the PI of the station currently active. In other cases, the position may be more effective. This way can be very effective, if a single line shall be selected in several Arrays (e.g., an entry in all telephone directories).

Notification on Arrays:
In case of a notification update, only those elements that have been changed are transmitted to the notified Controllers.
This is done by transmitting partial structures (elements or lines) as described above.

2.2.4.2.3 Function Class DynamicArray

The Arrays described above are optimized with respect to a high data volume. Navigation is based on the fixed sequence of elements in the Array (Position = PosX, PosY). The position will not be contained in the data field. In DynamicArrays, navigation is based on accessing a unique tag which identifies each element individually (tag is of data type Unsigned Word, replaces PosX, and is defined as the first parameter in the record). With the aid of these unique tags, it is possible to insert and delete elements at every position of the DynamicArray. Hence, the length of the DynamicArray may vary but the relative ordering is kept after inserting and deleting of elements.
DynamicArrays are defined as follows:

```
DynamicArray = Array of Record of {Tag, ...}
```

For function class DynamicArray, the messages are defined as follows:

OPType	Parameter
Set	**Tag**, **PosY**, Data
Get	Tag, PosY
SetGet	Tag, PosY, Data
Increment	Tag, PosY, NSteps
Decrement	Tag, PosY, NSteps
GetInterface	
Status	Tag, PosY, Data
Interface	Refer to section 2.2.4.2.2 on page 89
Error	ErrorCode, ErrorInfo

Tag	Unsigned Word	=	0x0000	all lines
		<>	0x0000	one special line
PosY	Unsigned Byte	<>	0x00	one special column (only if Tag <> 0x0000)
		=	0x01	not allowed, no access to Tag

The Tag belongs to the data field. This means that it is returned at the start of every line. PosY = 0x01 denotes the Tag. With respect to consistency, accesses to a column are not reasonable. The last line in a DynamicArray indicates the end. It starts with Tag 0xFFFF and contains dummy data. This line is included within the NMax counter.

Examples for positioning:

1. Array of Record of {Tag, El1, El2, El3}
2. Tag = 0x0000 and PosY = 0x00
3. Tag = 0x2006 and PosY = 0x00
4. Tag = 0x6389 and PosY = 3

(1)

Tag	El1	El2	El3
0356			
3467			
3624			
2006			
0101			
6389			
0900			
3581			
9023			
FFFF			

(2)

Tag	El1	El2	El3
0356			
3467			
3624			
2006			
0101			
6389			
0900			
3581			
9023			
FFFF			

(3)

Tag	El1	El2	El3
0356			
3467			
3624			
2006			
0101			
6389			
0900			
3581			
9023			
FFFF			

(4)

Tag	El1	El2	El3
0356			
3467			
3624			
2006			
0101			
6389			
0900			
3581			
9023			
FFFF			

Editing In DynamicArrays:

As with simple Arrays, data contents can be modified by using Set. In many cases, this is sufficient for DynamicArrays as well, especially if the inserting and deleting of lines is done within the Slave only. If the inserting and deleting of lines is done by the Controller as well, more complex editing functions are required. They will be defined as separate methods.
Below there are two examples which are defined in a way that they can be applied to several DynamicArrays (FktIDs), for example, several telephone directories. There is no need for an individual instance per Array. These functions will be placed in the range of Coordination (0x000...0x1FF).

By DynArrayIns (FktID=0x080), a number Quantity (Unsigned Word) of Array elements (entire lines) will be inserted in DynamicArray FktID. The lines will be inserted after that line containing Tag. A special case is inserting lines before the first line. In this case, Tag 0x0000 is used. The data contents of the lines to be inserted will be transferred as Data.

```
DynArrayIns.StartResultAck (SenderHandle, FktID, Tag, Quantity, Data)
```

Examples:

- **`DynArrayIns.StartResultAck (SenderHandle, FktID, 0000, 5, Data)`**
  Inserts 5 lines at the beginning of the DynamicArray
- **`DynArrayIns.StartResultAck (SenderHandle, FktID, 8795, 1, Data)`**
  Inserts 1 line after the line containing Tag 0x8795

DynArrayDel (FktID=0x081) deletes a number Quantity (Unsigned Word) of Array elements (entire lines). This is performed starting at the element containing Tag, which is included within deletion.

```
DynArrayDel.StartResultAck (SenderHandle, FktID, Tag, Quantity)
```

Examples:

- **`DynArrayDel.StartResultAck (SenderHandle, FktID, 0000, FFFF)`**
  Deleting of entire Array
- **`DynArrayDel.StartResultAck (SenderHandle, FktID, 8795, FFFF)`**
  Deleting of entire Array starting at line containing Tag 0x8795
- **`DynArrayDel.StartResultAck (SenderHandle, FktID, 8795, 0001)`**
  Deleting of the line containing Tag 0x8795
- **`DynArrayDel.StartResultAck (SenderHandle, FktID, 8795, 0000)`**
  No deleting

Notification on DynamicArrays

Notification on DynamicArrays is done in the same way as notification on Arrays. In contrast to Arrays the length of DynamicArrays is not fixed. If the length of a DynamicArray changes or a Tag value is changed (one entry replaced by another), the complete DynamicArray is transferred to the notified Controllers.

2.2.4.2.4 Function Class LongArray

A Slave transfers the Arrays and DynamicArrays (as described above) to the registered Controller using Shadows. In case of changes, the Shadows then will be updated. In case of big Arrays that are changed very often, this may not be practicable any longer (amount of memory in Controller, transmission time, bus load). Here another model – LongArray – must be applied. Where to place the boundary between LongArray and DynamicArray is a matter of an individual decision.
The class LongArray consists of a function MotherArray and one or more ArrayWindow functions. It is possible to generate instances of ArrayWindows dynamically. An ArrayWindow represents an extract, a window to the MotherArray. Function class LongArray, therefore, requires a minimum of two functions.

The Tag belongs to the data field. This means that it is returned at the start of every line. PosY = 0x01 denotes the Tag. With respect to consistency, accesses to a column are not reasonable. The last line in a MotherArray starts with Tag 0xFFFF and contains dummy data. The tag 0x000 is reserved to the Controller for initialization of the ArrayWindow. Therefore, it must not be used in the MotherArray.

2.2.4.2.4.1 MotherArray

The MotherArray is structured like a function of class DynamicArray but handling of MotherArrays is independent of any other Array.

The main difference compared to other Arrays is that the MotherArray is not controlled and viewed directly but via one or more different functions. Below there is an example of a MotherArray as Array of Record of {Tag, Character, Number}:

Tag	El 1	El 2
6243	a	01
2100	b	02
5428	c	03
0101	d	04
3245	e	05
4562	f	06
0012	g	07
5342	h	08
9473	i	09
9343	j	0A
8367	k	0B
3752	l	0C
7698	m	0D
	.	
	.	
	.	
6354	x	1E
3425	y	1F
1045	z	20
FFFF	FF	FF

2.2.4.2.4.2 ArrayWindow

The ArrayWindow represents a part of the MotherArray. One main difference to other function classes is that it is not useful to instantiate an ArrayWindow in a static way (via the FBlock Specification). In other function classes, where functions are instantiated in a static way, those functions describe fixed properties and methods of the Slave. Their state is identical for all Controllers. With respect to its status, an ArrayWindow is strongly bound to a Controller.
There must be an individual ArrayWindow for each Controller. It is possible that several HMIs have individual ArrayWindows to an address directory (MotherArray), which have different size and position.
Functions of class ArrayWindow are instantiated dynamically at runtime. Therefore, an FBlock (that has a MotherArray, which shall be accessed by ArrayWindows) must provide a method CreateArrayWindow for instantiation and a method DestroyArrayWindow (both of class Unclassified Method). The FktIDArrayWindow is used as instance handle, which is transferred from Slave to Controller during instantiation.

Function	OPTypes	Parameter
CreateArrayWindow	StartResultAck	**SenderHandle**, **FktIDMotherArray**, **PositionTag**, **WindowSize**
	ResultAck	SenderHandle, **FktIDArrayWindow**
	ErrorAck	SenderHandle, ErrorCode, ErrorInfo
DestroyArrayWindow	StartResultAck	SenderHandle, FktIDArrayWindow
	ResultAck	SenderHandle
	ErrorAck	SenderHandle, ErrorCode, ErrorInfo

SenderHandle	Unsigned Word	Unique identifier of a task.
FktIDMotherArray		FktID of the MotherArray. It is not dynamic, since MotherArray is a property of class DynamicArray of the Slave
FktIDArrayWindow		FktID of the ArrayWindow. It is generated dynamically and represents the object handle, which is transferred during instantiation. A range for such dynamically generated FktIDs is occupied in advance.
PositionTag	Unsigned Word	Top left corner of the ArrayWindow is positioned at PositionTag
WindowSize	Unsigned Byte	Number of elements contained by the ArrayWindow

The methods CreateArrayWindow and DestroyArrayWindow can instantiate and destroy ArrayWindows even of several MotherArrays. If, for example, in a telephone all telephone directories are available as MotherArrays, every HMI that is interested in a telephone directory may instantiate an ArrayWindow for the respective MotherArray. So it can be that, for example, three telephone directories may be watched by three ArrayWindows.

There are some situations, where Controller and Slaves destroy their ArrayWindows:

1. If modulated signal goes off, all instances of ArrayWindows are destroyed
2. If the FBlock containing the MotherArray is removed from the Central Registry.
3. Reception of Configuration.Status(NotOK)

A Controller is only allowed to destroy its own ArrayWindows.
Every Controller stores the position of its ArrayWindow with the help of the Tag of the first line inside the ArrayWindow. During CreateArrayWindow and by the help of MoveArrayWindow (SenderHandle, MovingMode, FktIDArrayWindow, Absolute, Tag), the window can be positioned again.

Upon creation an ArrayWindow is placed with its upper left corner to the Tag defined by the variable PositionTag. If the Controller has no information yet about the content of the MotherArray it can set PositionTag to 0x0000. That places the ArrayWindow to the top of the MotherArray.

A CreateArrayWindow with a PositionTag too close to the bottom of the MotherArray has the same effect as MoveArrayWindow.Bottom: It creates an ArrayWindow that contains the last WindowSize number of valid elements plus the last element with tag 0xFFFF.

If a Controller tries to position the ArrayWindow on a non-existent PositionTag, the ArrayWindow is positioned on the next greater tag. If there is no greater tag available, the previous lesser tag is used.
The status of an ArrayWindow is kept up to date in the Controller by using a Shadow. It is the Slave's task to keep the Shadow up to date. Here the notification mechanism of the Network Service cannot be used, since it is static. Notification for ArrayWindows is to be implemented at the application level. A creation of an ArrayWindow implies a notification on that ArrayWindow without the need of sending a Notification.Set message. For each ArrayWindow there is only one single Shadow, which is located in the Controller that has instantiated it. The DeviceID of the Controller is transferred to the Slave during instantiation, so there is no need to implement a special notification mechanism for registering the Controller.

A Slave always sends complete ArrayWindows (that does not necessarily mean full ArrayWindows) with parameter Tag = 0x0000 and PosY = 0x00 (this is also true for non-full ArrayWindows with less elements than WindowSize).
By using the ArrayWindow, editing the MotherArray can be done in the conventional way:

```
ArrayWindow.SetGet (Tag, PosY, Data)
```

Tag	El 1	El 2
6243	a	01
2100	b	02
5428	c	03
0101	d	04
3245	e	05
4562	f	06
0012	g	07
5342	h	08
9473	i	09
9343	j	0A
8367	k	0B
3752	l	0C
7698	m	0D
	.	
	.	
	.	
6354	x	1E
3425	y	1F
1045	z	20
FFF	FF	FF

0012	g	07
5342	h	08
9473	i	09
9343	j	0A
8367	k	0B

0012	g	07
5342	h	08
9473	i	B3
9343	j	0A
8367	k	0B

MotherArray ***ArrayWindow*** ***Set (9473, 03, B3)***

The Slave reacts on the Set operation by notifying all Controllers having an ArrayWindow on the edited column.

For inserting and deleting lines the following methods can be used:

These methods manipulate the ArrayWindow. The changes of the ArrayWindow are mirrored back to the MotherArray. Due to the fact that the MotherArray has changed its content, all ArrayWindows needing updates are getting an automatic update by the implicit notification, which was built up during instantiation of the ArrayWindows.

All changes to the content of the MotherArray, which result in changes to the content of an Array-Window or to the parameters AbsolutePosition or CurrentSize, lead to an update of the respective Shadows. In order to keep the communication load low, all changes to the MotherArray that do not lead to changes of an ArrayWindow or these parameters should not result in an update of the Shadows.

Function	OPType	Parameter
ArrayWindowIns	StartResultAck	SenderHandle, **FktIDArrayWindow**, **Tag**, **Quantity**, **InsertData**
	ErrorAck	SenderHandle, ErrorCode, ErrorInfo
	ProcessingAck	SenderHandle
	ResultAck	SenderHandle

SenderHandle	Unsigned Word	Unique identifier of a task.
Tag	Unsigned Word	Unique handle of a row (0xFFFF no valid value) after which the new lines are inserted 0x0000 indicates an insertion before the first line of the ArrayWindow
FktIDArrayWindow	Unsigned Word	FktID of ArrayWindow
Quantity	Unsigned Word	Number of rows
InsertData	Stream	Data to be inserted

Function	OPType	Parameter
ArrayWindowDel	StartResultAck	**SenderHandle**, **FktIDArrayWindow**, **Tag**, **Quantity**
	ErrorAck	SenderHandle, ErrorCode, ErrorInfo
	ProcessingAck	SenderHandle
	ResultAck	SenderHandle

SenderHandle	Unsigned Word	Unique identifier of a task.
Tag	Unsigned Word	Unique handle of a row (0xFFFF no valid value) after which the given number of lines is deleted
FktIDArrayWindow	Unsigned Word	FktID of ArrayWindow
Quantity	Unsigned Word	Number of rows

If any element of the ArrayWindow is deleted, the ArrayWindow keeps its position. All following elements move one position up and the resulting gap is filled up with the next element of the MotherArray.

Deleting elements within an ArrayWindow positioned to the bottom generates partially filled ArrayWindows because there are no elements in the MotherArray left to fill the gap. The ArrayWindow stays at its position. This exception is introduced to stabilize displays of the ArrayWindow to the user. The deletion can be caused by the Controller via command or by internal changes within the Slave.

If the last meaningful element of an ArrayWindow positioned at the bottom of the MotherArray is deleted, the ArrayWindow is shifted one window size up on the MotherArray. The result is the same as a Bottom operation. The ArrayWindow shows the WindowSize last elements of the MotherArray again.

Function	OPType	Parameter
ArrayWindow	Set	**Tag**, **PosY**, Data
	Get	Tag, PosY
	SetGet	Tag, PosY, Data
	Increment	Tag, PosY, Nsteps
	Decrement	Tag, PosY, Nsteps
	GetInterface	
	Status	Tag, PosY, **CurrentSize**, **AbsolutePosition**, Data
	Interface	Refer to section 2.2.4.2.2 on page 89
	Error	ErrorCode, ErrorInfo

Tag	Unsigned Word	= <>	0x0000 0x0000	all lines one special line
PosY	Unsigned Byte	<> =	0x00 0x01	one special column (only if Tag <> 0x0000) not allowed, no access to Tag
CurrentSize	Unsigned Word	Current size of the MotherArray The termination line with Tag 0xFFFF is not counted for CurrentSize. CurrentSize only indicates the number of meaningful lines		
AbsolutePosition	Unsigned Word	Absolute position of the ArrayWindow in the MotherArray. The value specifies the position of the top left cell in the MotherArray and the counting starts at 0.		

2.2.4.2.4.3 Positioning an ArrayWindow on a MotherArray

Since an ArrayWindow represents an extract of the MotherArray, it must be positioned on the MotherArray in an appropriate way. Therefore, two methods are defined. Method MoveArrayWindow is mandatory. An instance of MoveArrayWindow is used for all instances of ArrayWindows (FktID) of an FBlock.

```
MoveArrayWindow.StartAck ( Senderhandle, FktIDArrayWindow, MovingMode,
                           Number, Tag )
```

SenderHandle	Unsigned Word	Unique identifier of a task	
FktIDArrayWindow	Unsigned Word	FktID of the ArrayWindow to be moved	
Mode	Enum	00 01 02 03 04	Top Bottom Up Down Absolute
Number	Unsigned Word	Number of lines to move the window	
Tag	Unsigned Word	Unique handle of a row	

Top and Bottom:

Top and Bottom move the ArrayWindow to the start or the end of the MotherArray. The parameters Number and Tag are transferred as well, but they are not used in this mode.

If an ArrayWindow is positioned to Bottom, it contains the last WindowSize valid elements plus the last element with tag 0xFFFF. This means that in this case the ArrayWindow is one element bigger than in other cases.

Tag	El 1	El 2
6243	a	01
2100	b	02
5428	c	03
0101	d	04
3245	e	05
4562	f	06
0012	g	07
5342	h	08
9473	i	09
9343	j	0A
8367	k	0B
3752	l	0C
7698	m	0D
	.	
7589	v	0F
9643	w	1D
6354	x	1E
3425	y	1F
1045	z	20
FFFF	FF	FF

MotherArray

6243	a	01
2100	b	02
5428	c	03
0101	d	04
3245	e	05

0012	g	07
5342	h	08
9473	i	09
9343	j	0A
8367	k	0B

7589	v	0F
9643	w	1D
6354	x	1E
3425	y	1F
1045	z	20
FFFF	FF	FF

ArrayWindow

MoveArrayWindow.StartAck (SenderHandle, FktID, Top, xx, xxxx)

MoveArrayWindow.StartAck (SenderHandle, FktID, Bottom, xx, xxxx)

Up and Down:

Up and Down are used for relative movement of the ArrayWindow where the parameter Number (Unsigned Byte) defines the number of lines by which the ArrayWindow shall be moved. If the ArrayWindow is moved to a position that is outside of the MotherArray, it will be positioned at the closest point within the MotherArray. This means that it will be positioned at the Top or Bottom position depending on whether it was an Up or a Down command that tried to move it. No error will be reported.

The ArrayWindow stays at the Top or Bottom position.

A Controller cannot move the ArrayWindow out of the MotherArray's area - the ArrayWindow remains full. A MoveArrayWindow.Up command to an ArrayWindow positioned to the top of the MotherArray or a MoveArrayWindow. Down to an ArrayWindow positioned to the bottom result in no change to the position.

Tag	El 1	El 2
6243	a	01
2100	b	02
5428	c	03
0101	d	04
3245	e	05
4562	f	06
0012	g	07
5342	h	08
9473	i	09
9343	j	0A
8367	k	0B
3752	l	0C
7698	m	0D
	.	
	.	
	.	
6354	x	1E
3425	y	1F
1045	z	20
FFFF	FF	FF

MotherArray

0012	g	07
5342	h	08
9473	i	09
9343	j	0A
8367	k	0B

ArrayWindow

0101	d	04
3245	e	05
4562	f	06
0012	g	07
5342	h	08

9473	i	09
9343	j	0A
8367	k	0B
3752	l	0C
7698	m	0D

MoveArrayWindow.StartAck (SenderHandle, FktID, Up, 03, xxxx)

MoveArrayWindow.StartAck (SenderHandle, FktID, Down, 05, xxxx)

Absolute:

Absolute adjusts an ArrayWindow in a way that the first line contains the desired Tag. If the Tag is located too close to the end of the MotherArray, so that the ArrayWindow would exceed the valid range, the ArrayWindow will be placed as if Bottom had been used.

Tag	El 1	El 2
6243	a	01
2100	b	02
5428	c	03
0101	d	04
3245	e	05
4562	f	06
0012	g	07
5342	h	08
9473	i	09
9343	j	0A
8367	k	0B
3752	l	0C
7698	m	0D
	.	
	.	
	.	
6354	x	1E
3425	y	1F
1045	z	20
FFFF	FF	FF

MotherArray

0012	g	07
5342	h	08
9473	i	09
9343	j	0A
8367	k	0B

ArrayWindow

2100	b	02
5428	c	03
0101	d	04
3245	e	05
3245	f	06

MoveArrayWindow.StartAck (SenderHandle, FktID, Absolute, xx, 2100)

The second method SearchArrayWindow is optional. SearchArrayWindow provides a method to look for Searchstring in the MotherArray by means of an ArrayWindow (FktIDArrayWindow). Search is performed in that element of each line, which is specified by PosY:

```
SearchArrayWindow.StartResultAck (SenderHandle, FktIDArrayWindow, PosY,
                                  Searchstring)
```

Seeking starts from the first line of ArrayWindow and runs down to the end of the MotherArray. Then seeking continues automatically at the start of the MotherArray and ends at the first line of the ArrayWindow. In case of success, the first line of the ArrayWindow is positioned onto the first line of the MotherArray which contains Searchstring. In case of failure, an error is reported (ErrorCode 0x07 “parameter not available”).

If the Searchstring is closer to the bottom of the MotherArray than the size of the ArrayWindow, the ArrayWindow is placed as described in the section “MoveArrayWindow(Bottom)”.

Tag	El 1	El 2
6243	a	01
2100	b	02
5428	c	03
0101	d	04
3245	e	05
4562	f	06
0012	g	07
5342	h	08
9473	i	09
9343	j	0A
8367	k	0B
3752	l	0C
7698	m	0D
	.	
	.	
	.	
6354	x	1E
3425	y	1F
1045	z	20
FFFF	FF	FF

0012	g	07
5342	h	08
9473	i	09
9343	j	0A
8367	k	0B

5428	c	03
0101	d	04
3245	e	05
4562	f	06
0012	g	07

6354	x	1E
3425	y	1F
1045	z	20
FFF	FF	FF

MotherArray ***ArrayWindow*** ***SearchArrayWindow.StartResultAck (SenderHandle, FktID, 02, “c”)***
SearchArrayWindow.StartResultAck(SenderHandle, FktID, 02, “z”)

2.2.4.2.4.4 Re-Synchronization of ArrayWindows

Each device containing one or several LongArrays must offer the property LongArrayInfo to its Controllers. One instance of this property services all LongArrays present in the node. The purpose of this property is to enable Controllers to re-synchronize after a system error. By this property, Controllers can see if the ArrayWindows they created before still exist. It works like a normal Array except that it is only possible to perform a Get operation on it.

Note: *When multiple Controllers for the same LongArray exist in one device, the resynchronization has to be coordinated within the device.*

2.2.4.2.5 Function Class Map

The structure of the function class Map is similar to DynamicArray; however, no ordering of the elements may be assumed. It is optimized for Arrays with dynamic changes through both the Controller and the Slave. By not assuming an order of the lines in a Map, the communication overhead for notifications can be minimized.

Similar to the function class DynamicArray, lines of a Map are identified by a uniquely defined handle named *Tag* of data type Unsigned Word, which replaces *PosX*. Tags are not necessarily sorted and not necessarily continuous. Additionally, *Tag* is always the first element inside the record that defines the lines of the Array. The value 0xFFFF for *Tag* is reserved for indicating the end of the transmission of a whole Array in a status message (see below). The elements of the record are identified through *PosY*.

```
Map = Array of Record of {Tag, ...}
```

For the function class Map, the messages are defined as follows:

OPType	Parameter
Set	**Tag**, **PosY**, Data
Get	Tag, PosY
SetGet	Tag, PosY, Data
Increment	Tag, PosY, NSteps
Decrement	Tag, PosY, NSteps
GetInterface	
Status	Tag, PosY, {Data}
Interface	Refer to section **2.2.4.2.2** on page 89
Error	ErrorCode, ErrorInfo

Tag	Unsigned Word	=	0x0000	all lines
		<>	0x0000	one special line
PosY	Unsigned Byte	<>	0x00	one special column (only if Tag <> 0x0000)
		=	0x01	not allowed, no access to Tag

The Tag belongs to the data field and is returned at the start of every line. PosY = 0x01 denotes the Tag. With respect to consistency, accesses to a column are not reasonable. As with PosX, the value 0x0000 for Tag is reserved to indicate the whole Array. Because of the unordered nature of the function class Map, no last line with a Tag value of 0xFFFF needs to be stored. However, 0xFFFF is used to indicate the end of the transmission of the whole Array.

Examples for Accessing Elements of a Map:

Array of Record of {Tag, El1, El2, El3 }
Tag = 0x0000 and PosY = 0x00
Tag = 0x2006 and PosY = 0x00
Tag = 0x6389 and PosY = 0x03

(1)

Tag	El1	El2	El3
0356			
3467			
3624			
2006			
0101			
6389			
0900			
3581			
9023			
2712			

(2)

Tag	El1	El2	El3
0356			
3467			
3624			
2006			
0101			
6389			
0900			
3581			
9023			
2712			

(3)

Tag	El1	El2	El3
0356			
3467			
3624			
2006			
0101			
6389			
0900			
3581			
9023			
2712			

(4)

Tag	El1	El2	El3
0356			
3467			
3624			
2006			
0101			
6389			
0900			
3581			
9023			
2712			

Status Transmission for Function Class Map:

The status messages a Slave sends as a response to a Get request and for notifying the Controller of changes to a Map property contain the parameters Tag, PosY, and Data. Data contains one element, a whole line or several lines of the Array including the unique Tag of each line. The data field may be omitted to indicate the deletion of lines.

```
Map.Status ( Tag, PosY, {Data} )
```

The contents of a function class Map property are transmitted as follows:

1. Transmission of single entries:

   To transmit only one entry of a given line as a response to a corresponding Get request (PosY <> 0), the Slave sends a status message with the line's Tag, the position of the entry in PosY, and the contents of the entry in the parameter Data.

   The Slave may also use this type of transmission to notify the Controller when only one element of a line has changed.

2. Transmission of single lines:

   To transmit a whole line as a response to a Get request (PosY = 0), the Slave sends a status message containing the line's Tag, a PosY of 0x00, and the contents of the whole line in the parameter Data (including the line's Tag). Note that in this case the value of the parameter Tag and the line's Tag entry in the parameter Data have to be identical.

   The Slave may also use the single line transmission to notify the Controller of a change or insertion of a given line. In case of an insertion of a line the Slave sends a status message with the new line's Tag, a PosY of 0x00, and the contents of the line in Data to the Controller.

   If the Controller receives a message with a Tag already contained in its local copy, it must update the respective line accordingly. If it receives a notification message with an unknown Tag, it must add this line to its copy.

3. Combined transmission of changes and insertions:

   In a notification, multiple changes or insertions can be combined. In this case, the Tag is set to 0x0000 (to indicate that the complete Array is affected), PosY to 0x00, and the parameter Data contains a list with those lines that have changed or have been inserted. It is also allowed to notify single changes in this way. The Controller handles each item of Data separately as with single changes of insertions.

4. Transmission of whole Map property:

   If the whole contents of the Map property are transmitted, either as response to a corresponding Get request or as a notification, parameter Tag is set to 0x0000, parameter PosY to 0x00, and the last Tag in the parameter Data is the end Tag 0xFFFF. Tag 0xFFFF will be followed by dummy data. When receiving such a notification, the Controller must discard its local copy of the Map property and use the new, completely transmitted list.

5. Removal of line:

   To signal the removal of a certain line, the Slave sends a status message with only the Tag of the line and PosY to the Controller. To signal the deletion, parameter Data is not sent; parameter PosY is set to 0x00. If the Controller receives such a notification, it deletes the line with the transmitted Tag from its local copy.

The notification behavior for the function class Map is illustrated by the following examples:

Example 1: Notification of added line[1]

old list

Tag	El1	El2	El3
0356	011	012	013
3467	021	022	023

notification:
Tag = 0x3624, PosY = 0x00
Data =

Tag	El1	El2	El3
3624	031	032	033

new list

Tag	El1	El2	El3
0356	011	012	013
3467	021	022	023
3624	031	032	033

Example 2: Notification of modified line (using single line transmission)

old list

Tag	El1	El2	El3
0356	011	012	013
3467	021	022	023
3624	031	032	033

notification:
Tag = 0x3467, PosY = 0x00
Data =

Tag	El1	El2	El3
3467	N01	N02	N03

new list

Tag	El1	El2	El3
0356	011	012	013
3467	N01	N02	N03
3624	031	032	033

[1] Note that because of the unordered nature of the function class Map, the line could be inserted at any point in the array.

Example 3: Notification of multiple modifications

old list

Tag	El1	El2	El3
0356	011	012	013
3467	AAA	AAA	AAA
3624	031	032	033

notification:
Tag = 0x0000, PosY = 0x00
Data =

Tag	El1	El2	El3
3467	N11	N12	N13
2006	041	042	043

new list

Tag	El1	El2	El3
0356	011	012	013
3467	N11	N12	N13
3624	031	032	033
2006	041	042	043

Example 4: Notification with transmission of whole Array[1]

old list

Tag	El1	El2	El3
0356	011	012	013
3467	BBB	BBB	BBB
3624	031	032	033
2006	041	042	043

notification:
Tag = 0x0000, PosY = 0x00
Data =

Tag	El1	El2	El3
3624	031	032	033
2006	N21	N22	N23
0101	051	052	053
FFFF	d/c	d/c	d/c

new list

Tag	El1	El2	El3
3624	031	032	033
2006	N21	N22	N23
0101	051	052	053

Example 5: Deletion of line

old list

Tag	El1	El2	El3
3624	031	032	033
2006	N21	N22	N23
0101	051	052	053

notification:
Tag = 0x2006, PosY = 0x00

new list

Tag	El1	El2	El3
3624	031	032	033
0101	051	052	053

Editing in Function Class Map:

As in case of simple Arrays, data contents of a property of function class Map can be modified by using the operation types Set or SetGet in a similar way. However, similar to function class DynamicArray, more complex editing functions are required if insertion and deletion of lines is done by the Controller. These will be defined as separate methods in the coordination range (0x000 … 0x1FF) and can be applied to different Maps of an FBlock indicated by their FktIDs.

Using the method MapIns (FktID = 0x082), a number of Array elements (entire lines) will be inserted in the Map with the given FktID. Because the elements of a Map are not ordered, no given position for the insertion can be specified. The number of Array elements to insert is given in the parameter Quantity of data type Unsigned Word. The data contents of the lines to be inserted will be transferred in the parameter Data of type Stream. Because the Slave is responsible for assigning Tags, the Tag values in Data must be ignored (to avoid misunderstandings, these values should be set to 0xFFFF).

In case of using StartResultAck and if the insertion of the elements has been successful, the assigned Tags will be returned in the parameter TagList of the ResultAck message. The items in TagList must have the same order as the items in the parameter Data of the corresponding StartResultAck request.

```
MapIns.StartAck (SenderHandle, FktID, Quantity, Data )
MapIns.StartResultAck (SenderHandle, FktID, Quantity, Data )
MapIns.ResultAck (SenderHandle, FktID, Quantity, TagList )
```

[1] d/c (don't care) describes data that is to be ignored.

Examples:

```
Controller -> Slave: MapIns.StartAck (
       SenderHandle,  FktID, 0x0002, 0xFFFF EI1 EI2 EI3 0xFFFF EI1 EI2 EI3 )
```

Inserts two lines into the specified Map property.

```
Controller -> Slave: FBlock.MapIns.StartResultAck (
       SenderHandle,  FktID, 0x0002, 0xFFFF EI1 EI2 EI3 0xFFFF EI1 EI2 EI3 )
Slave -> Controller: FBlock.MapIns.ResultAck (
       SenderHandle,  FktID, 0x0002, 0x4312 0x4834 )
```

Two lines have been inserted into the specified Map property with Tags 0x4312 and 0x4834.

The method MapDel (FktID = 0x083) deletes a number of Array elements (entire lines) from the Map with the given FktID. The number of elements to delete is described through the parameter Quantity of data type Unsigned Word. The Tags of lines to be deleted are contained in the parameter TagList of type Stream. Because the elements in a Map are not ordered, no deletion of ranges is possible. However, using a Quantity of 0xFFFF and an empty TagList will delete the whole Map.

If the method MapDel is called with the OPType StartResultAck and if the deletion of the lines is successful, the given FktID is returned together with the number and the tags of the deleted lines in the parameters Quantity and TagList.

```
MapDel.StartAck (SenderHandle, FktID, Quantity, {TagList} )
MapDel.StartResultAck (SenderHandle, FktID, Quantity, {TagList} )
MapDel.ResultAck (SenderHandle, FktID, Quantity, {TagList} )
```

Examples:

```
Controller -> Slave: FBlock.MapDel.StartAck (
                          SenderHandle, FktID, 0x0001, 0x0101 )
```

Deletes the line with Tag 0x0101.

```
Controller -> Slave: FBlock.MapDel.StartAck (
                          SenderHandle,  FktID, 0x0002, 0x3581 0x2006 )
```

Deletes the lines with Tags 0x3581 and 0x2006.

```
Controller -> Slave: FBlock.MapDel.StartAck (
                          SenderHandle,  FktID, 0xFFFF )
```

Deletes the entire Array.

2.2.4.2.6 Function Class Sequence Property

The Sequence Property function class has a number of characteristics, which distinguish it from the Unclassified Property:

- OPType Status is mandatory and must have at least two parameters.
- If the OPTypes Set and/or SetGet are defined, the parameter list has to be identical to OPType Status.
- OPType Get must not have any parameters.

Note: *The individual parameters of the sequence do not have to be of the same type, that is, no repetition of one particular type is necessary.*

OPType	Parameters
Set	<**Parameter**>, <Parameter>{, <Parameter>}
Get	
SetGet	<Parameter>, <Parameter>{, <Parameter>}
GetInterface	
Status	<Parameter>, <Parameter>{, <Parameter>}
Interface	Flags, Class, Name, **NElements, IntDesc1, IntDesc2...**
Error	ErrorCode, ErrorInfo

Parameter:	Boolean BitField Enum Unsigned Byte Signed Byte Unsigned Word Signed Word Unsigned Long Signed Long String Stream[1] Classified Stream Short Stream	Parameter in the sequence.
NElements:	Unsigned Byte	Number of elements in function class Sequence Property

IntDescX are the interface descriptions of the single elements. Depending on the data type, one of the interface descriptions defined for the respective class can be inserted. Please note that in case of elements, parameter Flags is not available. For parameter OPTypes only Set, Get, SetGet, Status, and Error can be used.

[1] If data type Stream is used, the corresponding parameter must be the last parameter.

2.2.4.3 Function Classes for Methods

For methods there are only two function classes, since methods may differ significantly with respect to the parameters transferred during StartAck and ResultAck (in opposite to properties). Methods that do not belong to these classes belong to class “Unclassified Method”. They must be defined in a specific way.

Function Class	Explanation
Trigger Method	This class of methods is used to trigger something. They have no parameters.
Sequence Method	This class of methods has a number of parameters. An interface description can be generated according to section 2.2.4.3.2.
Unclassified Method	Unclassified Methods cannot be matched to any other function class in a meaningful manner.

Table 2-12: Classes of functions for a method.

2.2.4.3.1 Function Class Trigger Method

There are no parameters in case of Start/StartResult and it does not return parameters in case of Result or Processing.

OPType	Parameters
Start	
StartResult	
GetInterface	
StartResultAck	**SenderHandle**
StartAck	SenderHandle
ErrorAck	SenderHandle, ErrorCode, ErrorInfo
ProcessingAck	SenderHandle
Processing	
Result	
ResultAck	SenderHandle
Interface	Flags, Class, OPTypes, Name
Error	ErrorCode, ErrorInfo

SenderHandle: Unsigned Word Unique identifier of a task.

2.2.4.3.2 Function Class Sequence Method

In contrast to function class Sequence Property (see 2.2.4.2.6), the parameter lists for OPTypes StartAck, StartResultAck, and ResultAck do not have to be identical. However, at least one parameter besides SenderHandle is required.

Note: *The individual parameters of the sequence do not have to be of the same type; that is, no repetition of one particular type is necessary.*

OPType	Parameters
Start	<**Parameter**>{, <Parameter>}
Abort	
StartResult	<Parameter>{, <Parameter>}
GetInterface	
StartResultAck	**SenderHandle**, <Parameter>{, <Parameter>}
AbortAck	SenderHandle
StartAck	SenderHandle, <Parameter>{, <Parameter>}
ErrorAck	SenderHandle, ErrorCode, ErrorInfo
ProcessingAck	SenderHandle
Processing	
Result	<Parameter>{, <Parameter>}
ResultAck	SenderHandle, <Parameter>{, <Parameter>}
Interface	Flags, Class, Name, **NElements, IntDesc1, InDesc2, …**
Error	ErrorCode, ErrorInfo

Parameter:	Boolean BitField Enum Unsigned Byte Signed Byte Unsigned Word Signed Word Unsigned Long Signed Long String Stream[1] Classified Stream Short Stream	Parameter in the sequence.
SenderHandle:	Unsigned Word	Unique identifier of a task.
NElements:	Unsigned Byte	Number of elements in function class Sequence Method.

IntDescX are the interface descriptions of the single elements. Depending on the data type, one of the interface descriptions defined for the respective class can be inserted. Please note that in case of elements, parameter Flags is not available.

[1] If data type Stream is used, the corresponding parameter must be the last parameter.

2.2.5 Handling Message Notification

In many cases, HMIs and Controllers must get information about changes of properties in FBlocks. To avoid polling, events for automatic notification are defined. Such events must often be sent to several devices (e.g., two HMIs). Consequently, a Notification Matrix is implemented in every FBlock. The devices that should be notified of changes to the status of a property are registered in this matrix.
Only properties can be admitted to the Notification Matrix!

Entry	Fkt 1	Fkt 2	Fkt 3	Fkt 4	Fkt 5
DeviceID1	x	x	x	x	x
DeviceID2		x		x	
Free for entry					
Free for entry					
Free for entry					
Free for entry					

Table 2-13: Notification Matrix (x = notification activated)

The size of a Notification Matrix depends on the FBlock, on the number of properties, and on the number of device entries, each of which must be registered individually.

When taking into consideration that a DeviceID has 16bits, an FktID has 12bits, and that in some FBlocks possibly all 64 possible nodes of the network must be registered, the Notification Matrix may be very big. Nevertheless, the following subjects should be kept in mind:

- The Notification Matrix is only a model. It does not dictate the software implementation method.
- In most cases it is sufficient if the Notification Matrix has only a few entries.
- Group addresses are allowed as DeviceID in the Notification Matrix.
- Implementation may be done in more economical ways, for example, by pointers, in every function object that point to DeviceIDs.

For very simple FBlocks, for example, a CD changer, it is sufficient if the Notification Matrix provides only three entries for DeviceIDs. For example, by using a group address, all HMIs in the network can be notified of status changes.

Administration of the Notification Matrix is done via function Notification or by definition of the System Integrator.

If a Controller desires to register, or to remove registration, it sends the following command:

```
Controller -> Slave:
    FBlockID.InstID.Notification.Set
         (Control, DeviceID, FktID1, FktID2...)
```

Furthermore, it is up to the System Integrator or device supplier to define implicit notifications when necessary. This includes:

- the FBlock and property sending the event
- the DeviceID to which the event is sent

The implicit notification is set within the device like a normal notification on transition from System State NotOK to OK. So, the DeviceID that should be notified is registered in the Notification Matrix. Therefore, on this transition, the initial event is sent over MOST. Updates are sent upon changes of the property. The (implicitly set) notification can be requested by NotificationCheck and may be removed by FBlock.Notification.Set (Clear | ClearAll...) as with a notification set by function call. On transition from System State OK to NotOK, the implicit notification is cleared, as well.

When defining implicit notifications, the following restrictions have to be taken into account:

- The notification must only be defined for addresses, not depending on the dynamic address calculation (group addresses that do not depend on dynamic address calculation, broadcast address, static address, debug address, node position address of TimingMaster).
- After transition from System State NotOK to OK, the Controller receives the status reports of all functions which are activated as events.

A Slave should always be prepared to receive status messages, regardless of whether it implements a shadow FBlock. Also the reception of a status message when not being initialized must not confuse the shadow FBlock.

The DeviceID of the Controller is transported at the start of the message, as described in section 2.2.1 on page 40, but in order to enter group addresses, the DeviceID is transmitted in the parameter field as well. Parameter Control specifies where the entry or deletion is done:

Control	Name	Comment
0x0	SetAll	Entry is done for all functions
0x1	SetFunction	Entry is done for the following functions
0x2	ClearAll	DeviceID of Controller is deleted for all functions
0x3	ClearFunction	DeviceID of Controller is deleted for the specified functions
Rest	Reserved	

Table 2-14: Parameter Control

In the table below, the messages with the different Control values for making entries in the Notification Matrix are listed together with the respective resulting entries.

Message	Entry	Fkt 1	Fkt 2	Fkt 3	Fkt 4	Fkt 5
Notification.Set (SetAll, DeviceID1)	**DeviceID1**	x	x	x	x	x
Notification.Set (SetFunction, DeviceID2, FktID2, FktID4)	**DeviceID2**		x		x	
	Free for entry					
	Free for entry					
	Free for entry					
	Free for entry					

Table 2-15: Messages with different Control values and the resulting entries in the Notification Matrix

After receiving a Notification.Set message, the Slave has to initiate the reply of the first status message of the registered functions within $t_{Property}$. After having sent the Notification.Set message, the Controller waits for the reception of any of the status reports of the registered functions for $t_{WaitForProperty}$. If a device registers for a property that has already been registered for this device, the report is sent as if the device had been registered for the first time. However, this does not generate a second entry in the Notification Matrix. This also applies to registering with group addresses.

Deleting entries is done in a similar way. Deletion of a not notified function shall not cause an error message.

If a Controller desires to read information from the Notification Matrix, it sends:

```
Controller -> Slave: FBlockID.InstID.Notification.Get (FktID)
```

In general, all “Report” OPTypes (Status, Error, and Interface) are notified. Status and (possibly) Error are reported spontaneously after registration. Interface is not reported directly after registration.

As an answer to this request, a list is returned that contains all DeviceIDs that activated the respective FktID:

```
Slave -> Controller: FBlockID.InstID.Notification.Status (FktID, DeviceID1,
                                                    DeviceID2,..., DeviceIDN)
```

If a device sends a status message based on a notification and the MOST Network Interface Controller indicates to the Network Service that the transmission failed due to an invalid target address, all entries for this target address shall be deleted from the Notification Matrix of the FBlock sending this status message. This applies to notifications that were set by the Notification function as well as implicit notifications that were defined by the System Integrator.

Please note that when using the notification mechanism with group addresses, the notification must be deleted if the message could not be transmitted to any node. As long as there are one or more nodes receiving the message, the notification must not be deleted. Furthermore, the notification will not be deleted when using the debug address 0x0FF0 to send the notification.

Error handling:

The Notification property reports one of the following error messages, if any error occurred:

- No more registration possible[1]
  If no more registering is possible, function Notification answers:

  ```
  Slave -> Controller: FBlockID.InstID.Notification.Error (0x20, 0x01)
  ```

- FBlock not registered in the Notification Service
  This happens when the corresponding FBlock is not registered in the Notification Service.

  ```
  Slave -> Controller: FBlockID.InstID.Notification.Error (0x20, 0x20)
  ```

- No more registration possible
  If no more registering is possible, function Notification answers:

  ```
  Slave -> Controller: FBlockID.InstID.Notification.Error (0x20, 0x21)
  ```

- Notification.Set rejected
  The corresponding properties (FktIDList) reject the “Notification.Set” command because of a Notification Matrix overflow or because the property is not registered in the Notification Service.

  ```
  Slave -> Controller: FBlockID.InstID.Notification.Error (0x20, 0x10,
  FktIDList)
  ```

- Notification.Get not possible
  On a received “Notification.Get” command, whenever the respective property is not registered in the Notification Service, the following is reported:

  ```
  Slave -> Controller: FBlockID.InstID.Notification.Error (0x07, 0x01, FktID)
  ```

[1] ErrorInfo 0x01 is equivalent to ErrorInfo 0x21. ErrorInfo 0x21 should be preferred.

- No valid values or property failure
  In case a Controller registers at a time where no valid values of the respective property are available or a property becomes temporarily unavailable, Notification sends the following message to all nodes that are registered for the respective property:

  `Slave -> Controller: FBlockID.InstID.FktId.Error (0x41)`

  Application Example:
  This ErrorCode could be used to indicate the failure of a sensor. If the sensor signal disappears, the Sensor function would report the error “Not available” (0x41). If the function has an implemented notification mechanism, the error is distributed to the registered Controllers.
  This message is also sent, in case a node registers for the property after the problem occurred. Failure of a whole FBlock is described in section 3.1.5.4.

Reactions on system events

- Configuration.Status(NotOK)
  After reception of NotOK (and every time the system is regarded in state NotOK, for example, after start up) the Notification Matrix is deleted.

- Configuration.Status(NewExt)
  After receiving Configuration.Status(NewExt), every device checks whether it has to notify itself for one or more properties of the new registered FBlocks.
  In that case, the device has to register itself using Notification.Set.
  Only those notifications have to be requested which are related to the new FBlocks. It is not necessary to rebuild other notifications

- Configuration.Status(Invalid)
  After receiving Configuration.Status(Invalid), every device checks whether it has notified itself for one or more properties in the deregistered FBlocks.
  In this case, the device has to react in an appropriate way to adjust its internal structure to the new situation.

3 Network Section

The Network Section comprises of Network Layer Service Specification and the Network Layer Protocol Specification. This includes network management, connection management, dynamic behavior, and error management.

The MOST Network Service provides all the basic functionality to operate a MOST system. It contains a comprehensive library of API functions to interface with the hardware and simplify use of MOST for the application.
The MOST Network Service offers a wide variety of functions for implementing applications. Some functions are mandatory for a MOST device. MOST Network Service provides a basic framework for a MOST device.

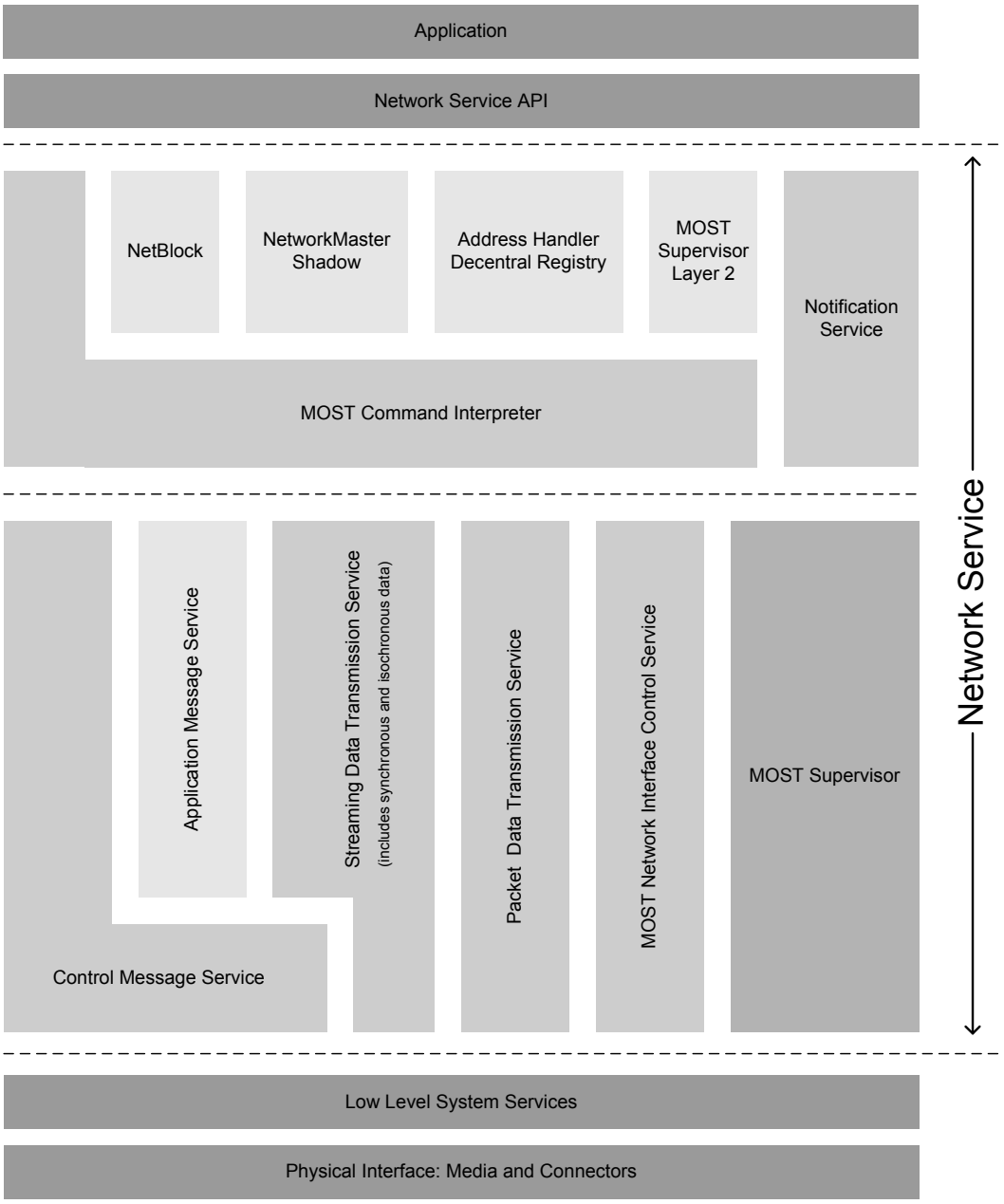

Figure 3-1: Application and Network Service of a MOST device

Note: *The MOST Supervisors combine a number of different responsibilites and are therefore not described as single components in this specification. The MOST Supervisor is, for example, responsible for initialization (3.1.3.1.1.1), Lock handling (3.1.2.2.2), NetInterface state transitions (3.1.2.2), and ring break diagnosis (3.1.4.1).*
The MOST Supervisor Layer 2 handles address generation (3.2.2) and System State changes within the NetInterface Normal Operation state (3.1.3.2).

3.1 Network Layer Service Specification

The Network Layer Service Specification defines the abstract interface between the Application Layer and the Network Layer.

3.1.1 MOST Data

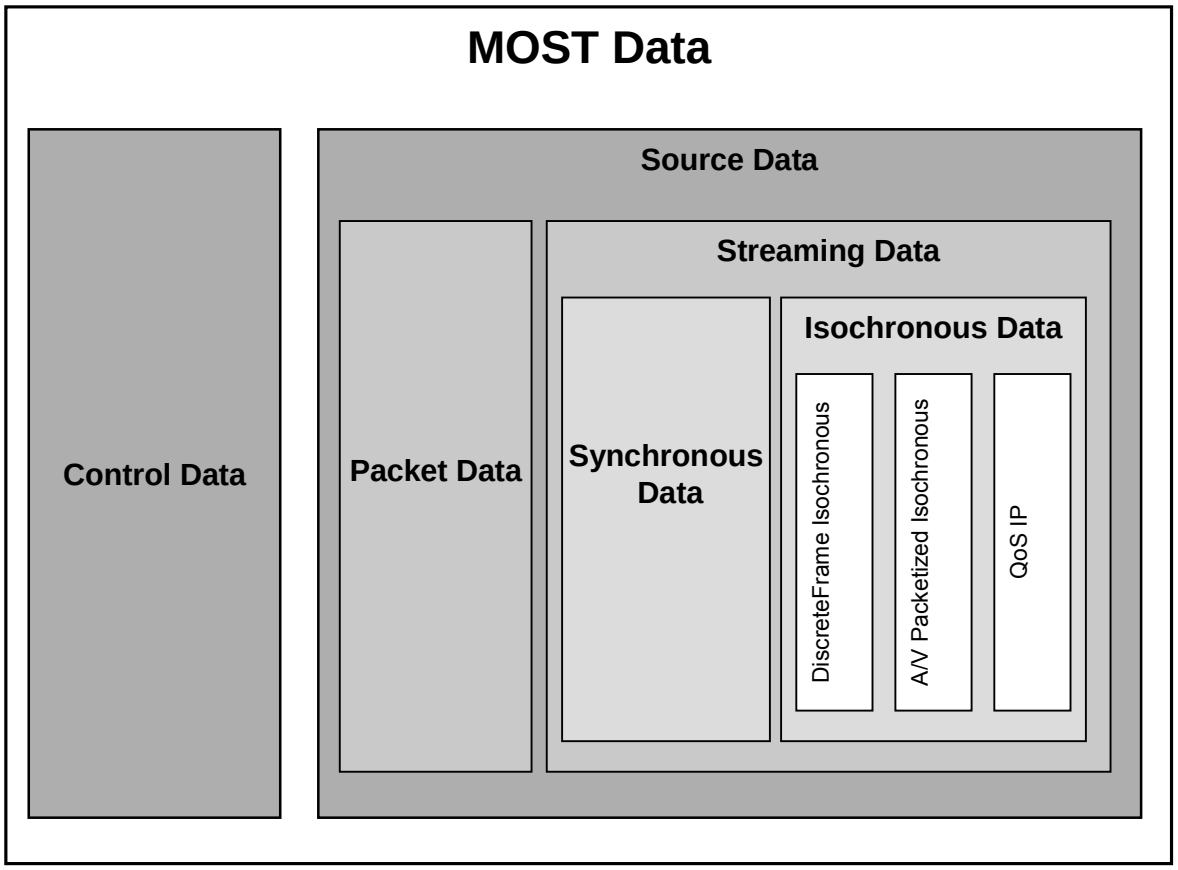

Figure 3-2: MOST data transport mechanisms

3.1.1.1 Control Data

Messages on the Control Channel provide control of applications.

The messages can be sent as single cast (logical or node position addressing), groupcast, or broadcast.

Arbitration is provided automatically by the MOST Network Interface Controller in case a node wants to send a message. In order to provide fair arbitration even at high bus loads, a double arbitration mechanism is used. This ensures that an access is not depending on the communication load of upstream devices and the priority is not depending on the network position.

Rejection of messages by the receiver is flagged and automatic retransmission is performed. The number of retries can be defined by the application software. If the maximum of retries is reached without success, a transmission error is indicated to the application.

3.1.1.2 Source Data

The MOST Network Interface Controller handles and converts a variety of local data formats and transmits that data over MOST. “Source data” refers to every kind of data that is not transmitted over the Control Channel: packet data and streaming data (see Figure 3-2: MOST data transport mechanisms).

3.1.1.3 Distinction between Source Data and Control Data

Depending on the kind of data and bandwidth, the MOST system provides different transmission procedures.

Telegrams for controlling devices or slow packet data are transmitted via the Control Channel of the MOST Network Interface Controller. For transmitting packet data of higher bandwidth, a packet-oriented data area is available. Streaming data, such as audio signals of a CD drive, can be transmitted directly in the streaming data area of the network. A more detailed description of the different data areas can be found in the sections below.

3.1.1.4 Differentiating Streaming Data and Packet Data

The number of streaming and packet data bytes is specified by the Boundary Descriptor value described in 3.2.8.1.

3.1.1.5 Synchronous Data

Streaming data connections are available for real-time data (synchronous data) such as audio/video or sensor data and eliminate the need for additional buffering in analog-to-digital converters (and digital-to-analog converters) or in single speed CD devices for audio and video.

Accessing this data is provided by time division multiplexing (TDM) and allocation of quasi-static physical connections for a certain period of time (e.g., while playing an audio source). The bandwidth for such a connection can be adjusted by allocating the required number of bytes.

3.1.1.6 Isochronous Data

Isochronous data is source data which is received or sent at a rate that is unrelated to the MOST system clock. Accessing this data is provided by time division multiplexing (TDM) and allocation of quasi-static physical channels for a certain period of time. The bandwidth for such a channel can be adjusted by allocating any number of bytes to one logical channel.

As with synchronous data, data on an isochronous channel is transmitted from a single source to one or more sinks. Therefore no arbitration has to be done.
The isochronous transmission class does not provide error detection (e.g., CRC). The receiving application has to cope with frames corrupted during transmission.

Three use cases are distinguished:

a) DiscreteFrame Isochronous (Streaming)
Isochronous streams containing data and additional timebase information in parallel. Typically the data is arranged in frames.

All PCM streams fall into this category. Here the PCM data is accompanied by a frame sync signal.

b) A/V Packetized Isochronous (Streaming)
Isochronous streams containing only data. Additional timebase information may be transported 'inside' the data. Typically the data is arranged in packets.

MPEG System Layer streams (e.g., MPEG2-TS) fall into this category. No frame sync signal is included; nevertheless the data contains multiple MPEG timestamps.

c) QoS IP (Streaming)
This mode is used for packet transmission which allows for full quality of service requirements.

Note: *For more information on isochronous data, refer to the MOST Stream Transmission Specification.*

3.1.1.7 Packet Data

Another time slot is available for packet data transport as required for burst-like data. In contrast to the Control Channel, the Packet Data Channel provides transmission of longer data packets. It also provides two different addressing mechanisms:

16 bit Target Address (classic MOST addressing)
48 bit Target Address (MAC addressing)

Access to this type of data is provided in a token ring manner. Each node has fair access to this channel and its bandwidth can be controlled using the Boundary Descriptor in a step of four bytes (quadlets).

The maximum packet length on the Packet Data Channel in 48 bit addressing mode is 1522 bytes. In this mode, each node must support the full packet length.
The maximum packet length on the Packet Data Channel in 16 bit addressing mode is 1532 bytes.
In the 16 bit and 48 bit addressing modes, data is transported byte aligned.

The data on this channel, consisting of target address, source address, and the payload (Data area) is CRC protected by the Packet Data Channel CRC. The packet data message is defined as follows.

Area	Size / bytes	Task
Administration	H	Data Link Layer Overhead (e.g., Arbitration, Length in bytes, Transmission Status, Addressing Type Identifier (16 bit or 48 bit mode))
	2	Target address
	2	Own address (Source address)
	4	Packet Data Channel CRC
Data	N	N = 6[1] … 1524

Table 3-1: Structure of packet data (16 bit addressing)

Area	Size / bytes	Task
Administration	H	Data Link Layer Overhead (e.g., Arbitration, Length in bytes, Transmission Status, Addressing Type Identifier (16 bit or 48 bit mode))
	6	Target address
	6	Own address (Source address)
	4	Packet Data Channel CRC = Ethernet FCS
Data	N	Data (containing Ethernet frame without destination, source and FCS) N = 0 … 1506

Table 3-2: Structure of packet data (48 bit addressing)

Since the packet data area is variable, it can take several frames to complete a message. The corresponding management such as arbitration and channel allocation is provided by the MOST Network Interface Controller.
A hardware Packet Data Channel CRC is provided. The Packet Data Channel CRC is calculated in the background and can be indicated in a register at the end of each packet data message.

Low-level retries are supported for both addressing modes. The number of low-level retries and the time between individual low-level retries are defined by the System Integrator.

[1] For information on the use of the first 6 bytes, please refer to section 3.2.6.1.1.

3.1.2 Dynamic Behavior of a Device

3.1.2.1 Overview

This section describes the dynamic behavior of the system — the states and state transitions of the system, with a special focus on network dynamics (i.e., the dynamics of the Network Interface of a device). The expression NetInterface stands for the entire communication section of a node, that is, the physical interface, the MOST Network Interface Controller, and the Network Service.

A MOST Node implements the NetBlock, FBlock EnhancedTestability, and Network Service.

For network analysis purposes, a certain node kind can be configured to operate in “listen only” mode. From the network view, these nodes are invisible.
“Listen only” nodes are typically not part of MOST systems in the field; they are only included during system design and for failure diagnosis.

A MOST device contains at least one MOST Node. If the device contains more than one node, it is called a multi-node device.
Generally, for each device, the Device Specification must define all the possible combinations of the states of the application section and the communication section. Particularly from the view of the network, there are three states that are mandatory for each device:

DevicePowerOff: Communication section is in state NetInterface Off.

1. **DeviceStandBy:** The logical function of the application is running, while peripherals with high power consumption such as drives are switched off. This state is reached after state DevicePowerOff. The communication section is in state NetInterface Normal Operation.
2. **DeviceNormalOperation:** The communication section, as well as the application, is in state normal operation.

The following description gives an impression of what may happen in the single states with respect to the communication and application sections.

DevicePowerOff:

- The application may be awakened (for example, by a timer), can check an external signal, and return to sleep mode without waking up the NetInterface. The device does not leave the mode.
- The application may be awakened, for example, by a timer, and can then wake up the NetInterface, and by that, the entire network. The device changes to state DeviceStandBy or state DeviceNormalOperation.
- The application may be awakened when detecting a modulated signal on the bus and then wakes the application during initialization phase. The device changes to state DeviceStandBy.

DeviceStandBy:

- If the application is used, or its peripherals are in use, the device changes to state DeviceNormalOperation.
- If the modulated signal on the bus is switched off, the device changes to state DevicePowerOff.

DeviceNormalOperation:

- If the modulated signal on the bus is switched off, the device changes to state DevicePowerOff.

3.1.2.2 NetInterface

Here, the states of a device are seen from the view of the NetInterface. Operations within the application of a device are not considered. Only the interfaces to the application are shown.
The figure below shows the states of the NetInterface and the events that lead to state transitions. In addition, the System State that corresponds to the NetInterface state is indicated.

The state NetInterface Diagnosis Result is optional.

The following sections explain the individual states.

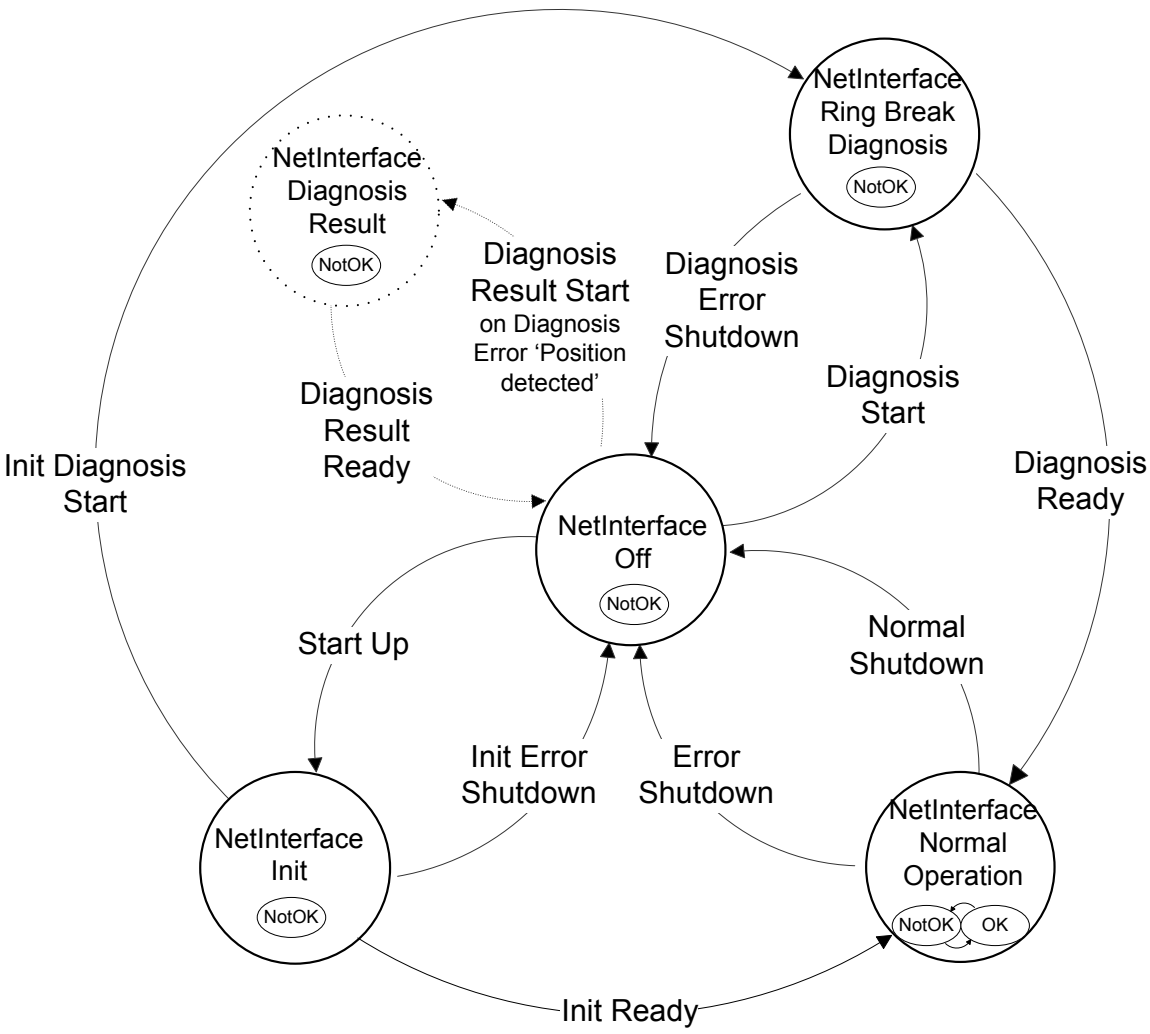

Figure 3-3: Overview of the states in NetInterface

3.1.2.2.1 NetInterface Off

In state NetInterface Off, the NetInterface is switched off from the view of the network. That means there is no modulated signal. The MOST Network Interface Controller does not necessarily need to be switched off since the application may still use function groups of it (e.g., local clock generation).

State NetInterface Off is left when one of the following events occurs:

Event	Transition to	Cause
Start Up	NetInterface Init	A NetInterface is activated either by a modulated signal at the receiving physical interface, by the application (hypothetical example: phone receives a call), or by a switch at the device.
Diagnosis Start	NetInterface Ring Break Diagnosis	A NetInterface is activated by connecting to power.
Diagnosis Result Start	NetInterface Diagnosis Result	Diagnosis error “Position detected”.

Table 3-3: Events in state NetInterface Off

3.1.2.2.2 NetInterface Init

In this state, NetInterface is initialized to the point where the MOST Network Interface Controller is able to communicate with other nodes.

When entering state NetInterface Init, the TimingMaster clears the System Lock Flag on Data Link Layer. This value is transferred to all MOST Network Interface Controllers. As soon as the TimingMaster recognizes a Stable Lock, it sets the System Lock Flag in the administrative area of the MOST frame.

This state is left when one of the following events occurs:

Event	Transition to	Cause
Init Ready	NetInterface Normal Operation	NetInterface is ready for communication (see below).
Init Error Shutdown	NetInterface Off	Error occurred during initialization (see below). The Shutdown Flag is not set and $t_{SSO\_Shutdown}$ does not apply.
Init Diagnosis Start	NetInterface Ring Break Diagnosis	A Diagnosis Start trigger is received.

Table 3-4: Events in state NetInterface Init

Causes for event Init Ready:

- In the TimingMaster device:
  Stable Lock (for a minimum time of t_{Lock}) was recognized. Lock is called stable if for a period of time t_{Lock} no unlock events occurred.
- In a TimingSlave device:
  Stable Lock was recognized and the System Lock Flag is set. Lock is called stable if for a period of time t_{Lock} no unlock events occurred.

Causes for event Init Error Shutdown:

- In the TimingMaster device:
  Timeout t_{Config} occurs before a Stable Lock can be recognized.
- In a waking TimingSlave device:
  Timeout t_{Config} occurs before a modulated signal is recognized.
- In a woken TimingSlave device:
  - Timeout t_{Config} expires before stable lock has occurred and a closed ring is recognized
  - The modulated signal was switched off again.

In a woken TimingSlave device, the bypass of the MOST Network Interface Controller is deactivated (opened) as soon as a short lock is recognized (i.e., the lock does not need to be stable for t_{Lock}). In case of a waking TimingSlave device and the TimingMaster device, the bypass is deactivated immediately after having entered this state (modulated signal at the output).

As soon as the initialization of the MOST Network Interface Controller starts, the logical node address has to be set to 0x0FFE.

The flow chart below shows the behavior in state NetInterface Init. A differentiation is made between Master and Slave. On this level, Master means TimingMaster and Slave means TimingSlave.

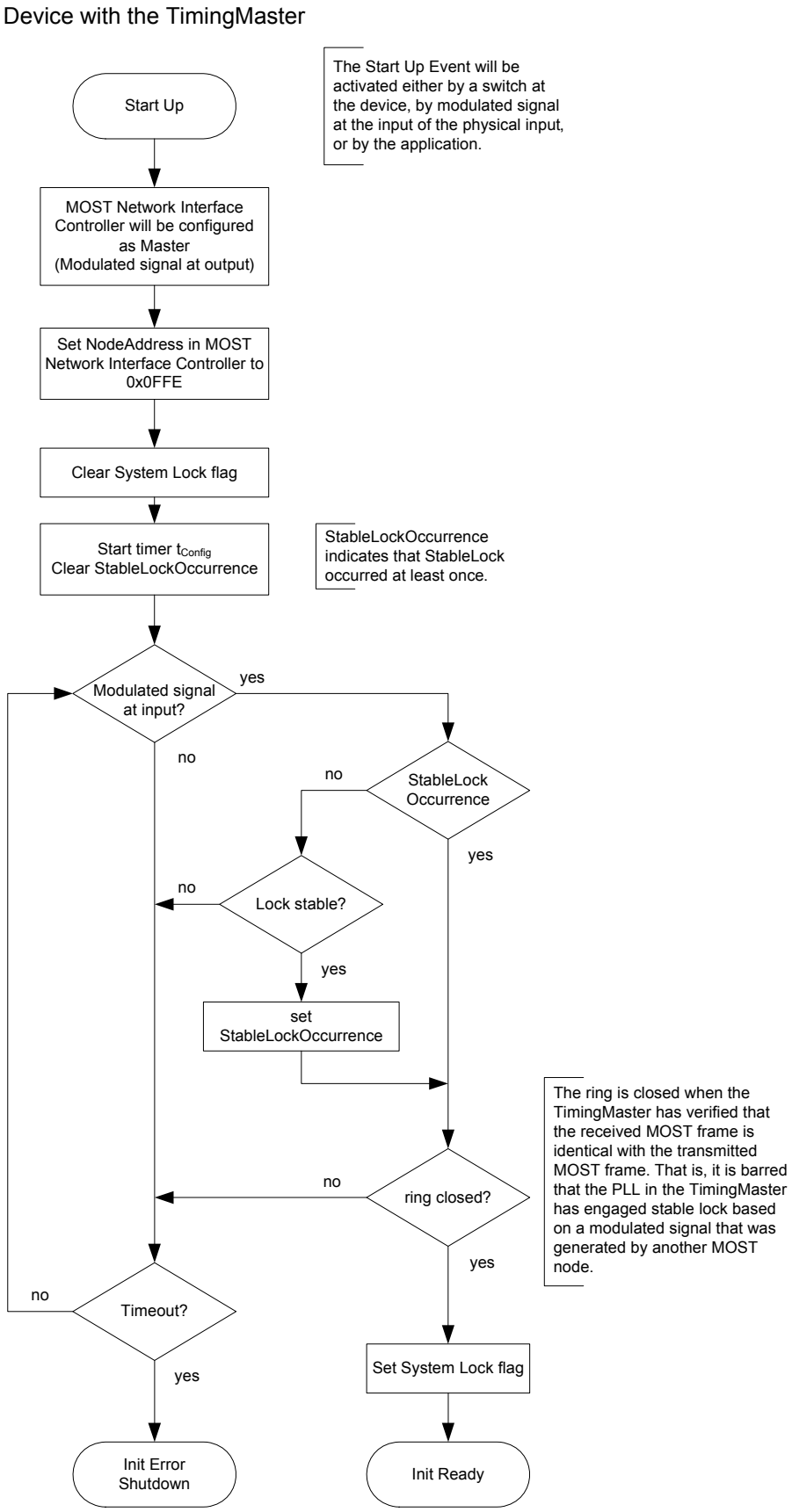

Figure 3-4: Behavior of a TimingMaster device in state NetInterface Init

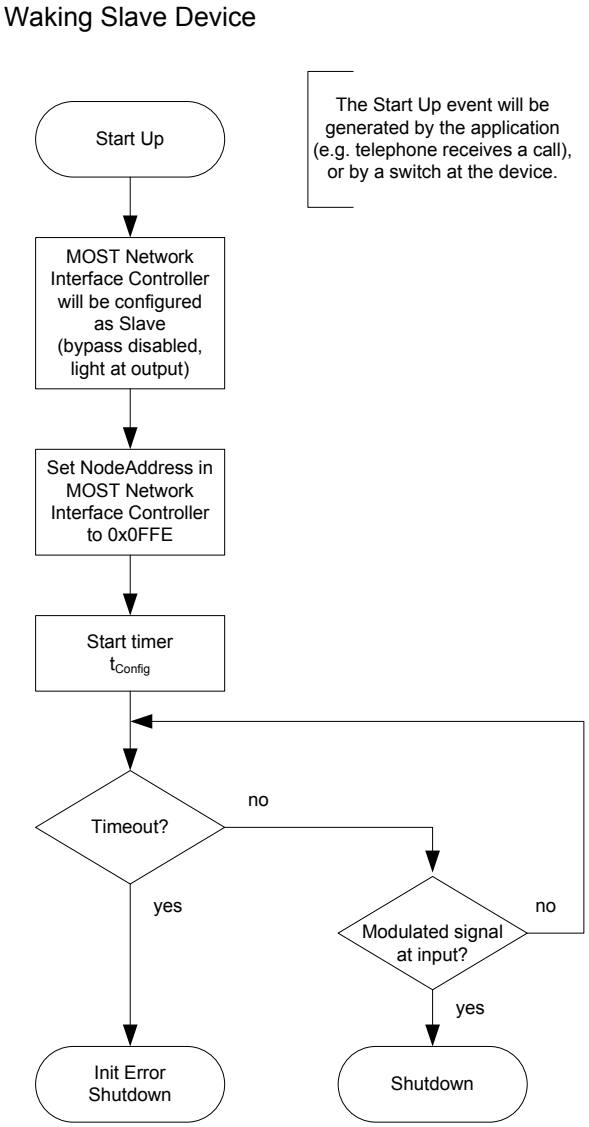

Figure 3-5: Behavior of a waking TimingSlave device in state NetInterface Init

After having woken the ring (a modulated signal generated by the TimingMaster is received at the input), the TimingSlave device goes to Shutdown[1]. From there it starts up as a standard TimingSlave device, woken by the TimingMaster.

[1] The “Shutdown” transition is not contained in *Figure 3-3: Overview of the states in NetInterface*. It leads to state NetInterface Off.

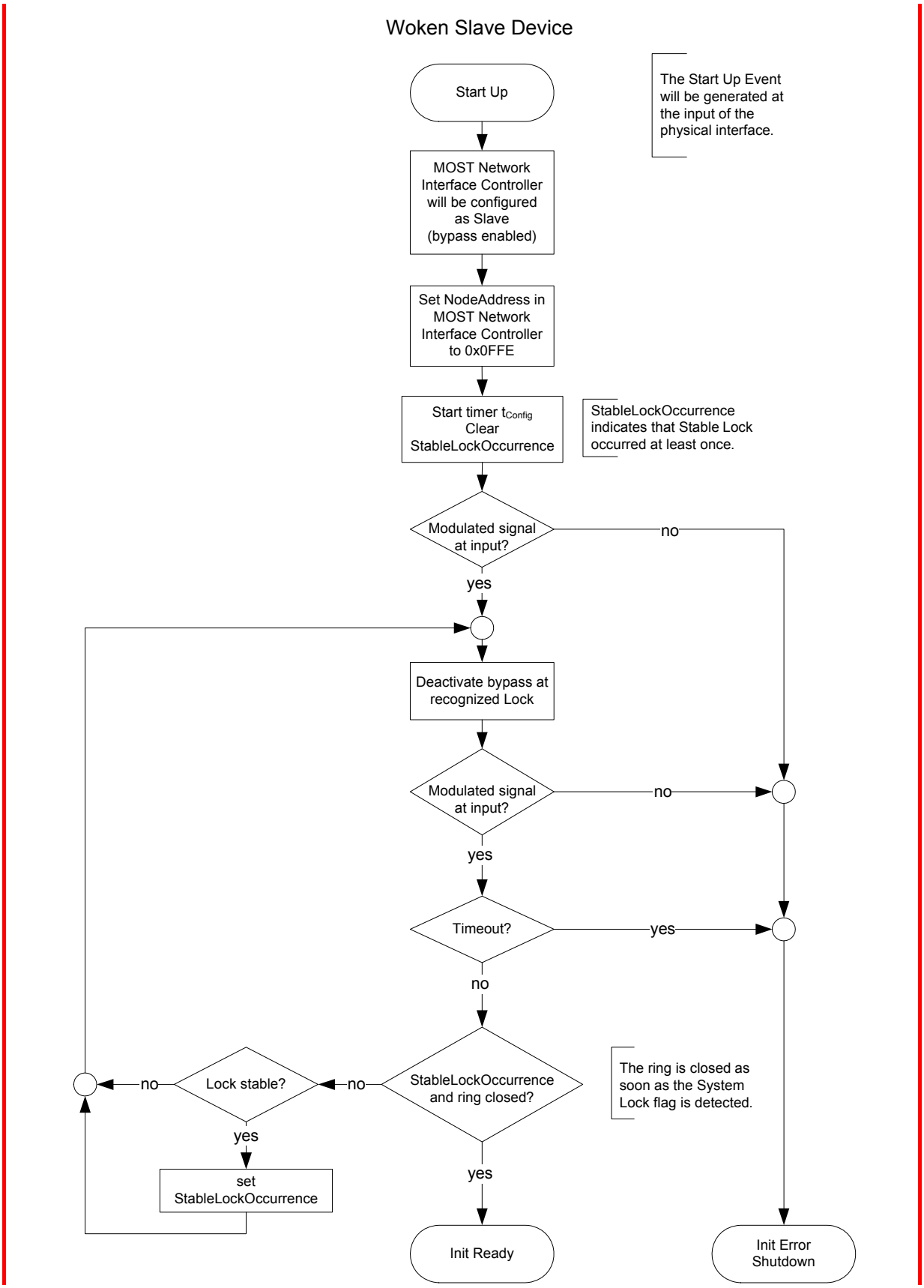

Figure 3-6: Behavior of a woken TimingSlave device in state NetInterface Init

3.1.2.2.3 NetInterface Normal Operation

This state is reached as soon as the initialization has reached a level where the MOST Network Interface Controller can start to communicate with other nodes in the network. When entering this state, the part of the application that is connected to the communication section is initialized.
The NetInterface Normal Operation state is also commonly referred to as NetOn state.

Examples for initializing a higher layer due to the Init Ready event:

- Check of system configuration and building of the Central Registry (refer to section 3.1.3.3).
- Setting of the logical node address and group address (refer to section 3.1.3).
- Initialization of the sending and receiving parts of the Network Service.

In certain circumstances, other application units are initialized earlier, independently from the state of the NetInterface.

Event	Transition to	Cause
Normal Shutdown	NetInterface Off	NetInterface will be deactivated by switching off the modulated signal.
Error Shutdown	NetInterface Off	NetInterface will be deactivated due to a Critical Unlock.
Init Ready	Report to an application	Entering state NetInterface Normal Operation

Table 3-5: Events in state NetInterface Normal Operation

The Normal Shutdown event is generated as soon as no modulated signal is recognized at the input.

In state NetInterface Normal Operation, the Network Service checks the lock state of the PLL of the MOST Network Interface Controller. If an unlock occurs, the application must be informed as soon as possible by an unlock event. Please refer to section 3.1.5.2 ‘Unlock’ on page 167 for more information regarding this type of error.

The flow chart below shows the behavior in state NetInterface Normal Operation:

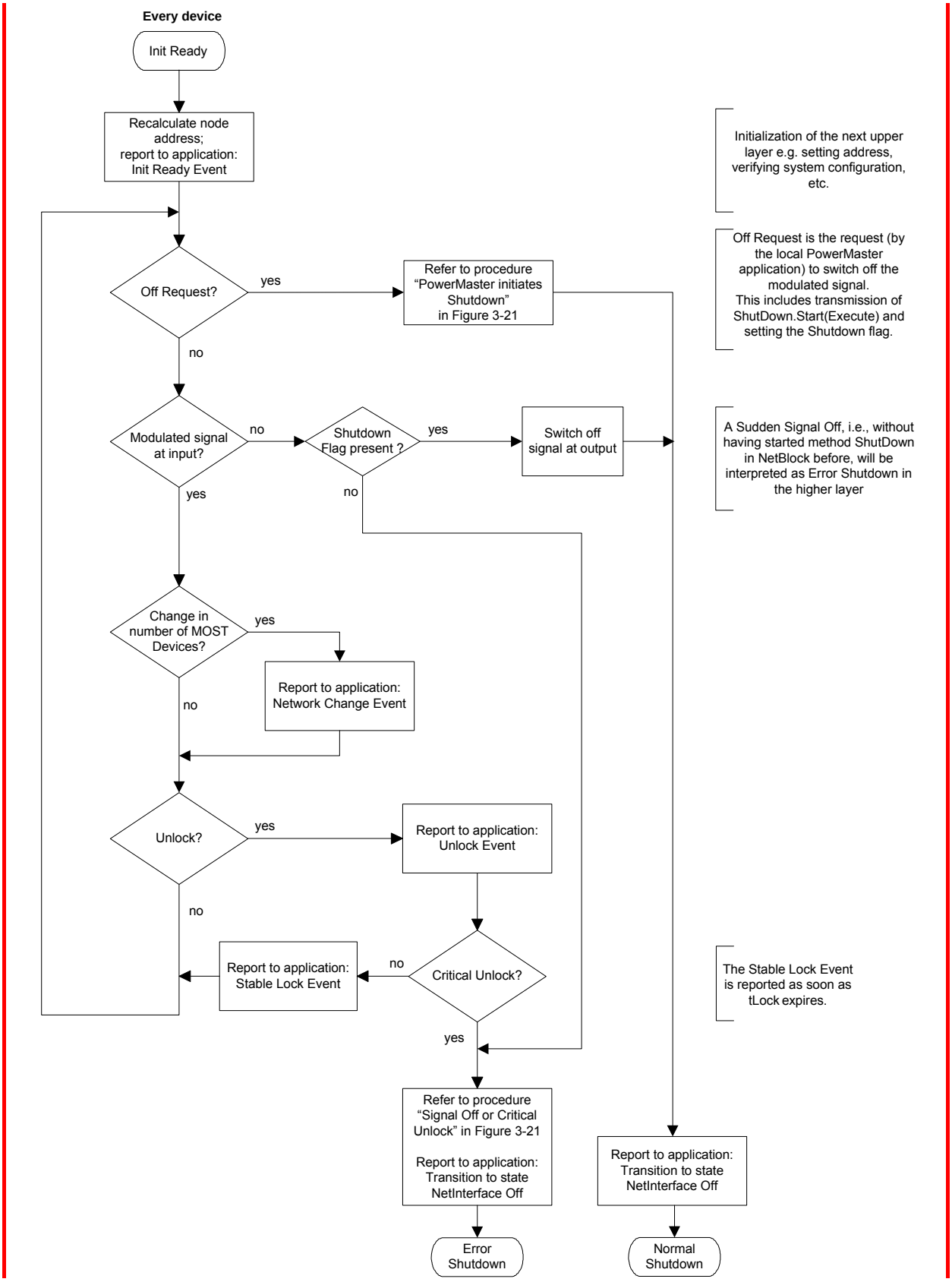

Figure 3-7: Behavior in state NetInterface Normal Operation

3.1.2.2.4 NetInterface Ring Break Diagnosis

Ring break diagnosis serves the purpose of localizing a fatal error in the network.

The ring break diagnosis process can be started by various triggers, which must be chosen and implemented by the System Integrator.

In state NetInterface Ring Break Diagnosis, a relative node position is determined in every device. This information can be used in case of a fatal error (ring break, attenuation—no lock possible, or defective device) to localize the error (see 3.1.4.1 Ring Break Diagnosis).

If there is no fatal error, the NetInterface immediately changes to state NetInterface Normal Operation.

In case of a Diagnosis Error Shutdown event, the position determined in each device describes the position relative to the device that was configured as TimingMaster at the end of ring break diagnosis (since there was no modulated signal at its input).

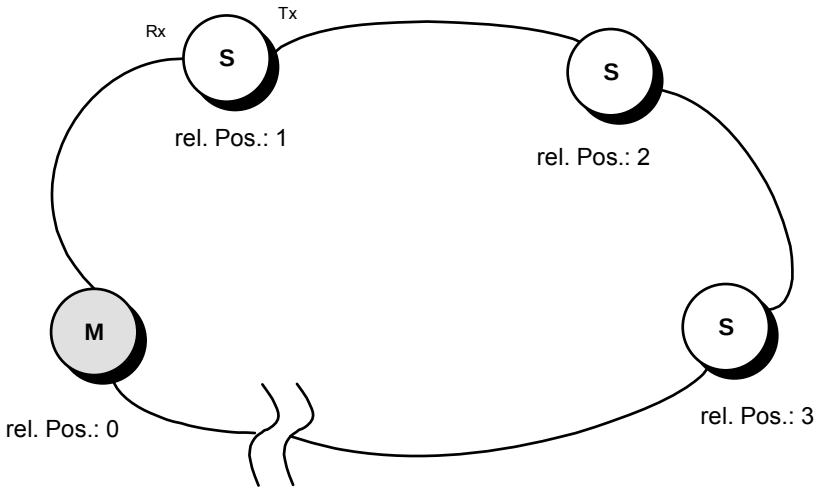

Figure 3-8: Localizing a fatal error with the help of ring break diagnosis.

Event	Transition to	Cause
Diagnosis Ready	NetInterface Normal Operation	No fatal error.
Diagnosis Error Shutdown	NetInterface Off	Fatal error (Ring break or defective device). The Shutdown Flag is not set and $t_{SSO\_Shutdown}$ does not apply.
Diagnosis Error Shutdown	NetInterface Diagnosis Result	Fatal error (Ring break or defective device) This transition is only available if the optional NetInterface Diagnosis Result state exists. The Shutdown Flag is not set and $t_{SSO\_Shutdown}$ does not apply.

Table 3-6: Events in state NetInterface Ring Break Diagnosis

3.1.2.2.5 NetInterface Diagnosis Result

This is an optional state.

Event	Transition to	Cause
Diagnosis Result Ready	NetInterface Off	Phase 3 of RBD (delivery of result) complete. The Shutdown Flag is not set and $t_{SSO\_Shutdown}$ does not apply.

Table 3-7: Events in state NetInterface Diagnosis Result

3.1.2.3 Power Management

Power management means that the administrative function, which is above the Network Service, wakes and shuts down the MOST network or specific devices. The power management is handled mainly by the PowerMaster, which uses NetBlock functions for this purpose.

3.1.2.3.1 Waking of the Network

Waking the network is done by switching on a modulated signal. In principle the network can be awakened by any node. The PowerMaster itself will usually wake the network, for example, when there is communication on the car's bus or based on the status of the vehicle (Clamp status).

A device must only wake the network when this is initiated by the application. Failure (e.g., supply voltage too low or too high) must not initiate waking of the network. Other solutions for waking up the network have been implemented as well, such as using a separate electrical wake-up line. It is up to the System Integrator to choose the preferred wake-up method. The process described here is independent of the wake-up method.

When an application wakes the network, it calls the respective routine in the Network Service, which switches on a modulated signal at the output of the device. Every node that recognizes the modulated signal at its input switches on a modulated signal at its output and initializes. In this way, the modulated signal travels from node to node until the entire network is awake.

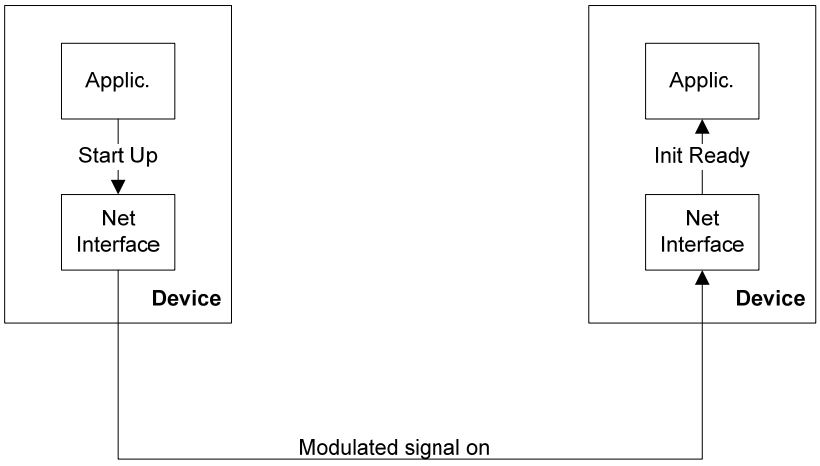

Figure 3-9: Example (2 devices) for waking of the MOST network via a modulated signal on the network

3.1.2.3.2 Network Shutdown

Switching off the network is done on lowest level by switching off the modulated signal. A device, which has switched off the modulated signal, must not switch it on again before $t_{Restart}$ occurred. This applies even if it recognizes a modulated signal at its input (modulated signal wake-up) or any other wake-up trigger.

Error Shutdown
All devices only switch off the modulated signal when certain errors occur (Critical Unlock, low voltage with reset). Refer to *3.1.5.2 Unlock* and *3.1.4.2 Detection of Sudden Signal Off and Critical Unlock*.

Normal Shutdown
In all other cases, only the PowerMaster switches off the network. For avoiding that devices have to save their status to persistent memory very often, the PowerMaster implements a shutdown procedure that has two stages. This procedure contains request and execution. For requesting, it starts method ShutDown with parameter Query in all NetBlocks of the system. This is one of the rare cases where a telegram is broadcasted.
After that, the PowerMaster waits for $t_{WaitSuspend}$ before it sends ShutDown.Start(Execute), using the blocking broadcast address, to announce the shutdown of the system. A device without any further need for communication does not respond on ShutDown.Start(Query).

The execution is announced by the PowerMaster by starting ShutDown.Start(Execute). By this function call, the shutdown process is started irrevocably. The devices do not reply to this call. They prepare for shutting down (saving status) and then wait for the modulated signal to be switched off.

The PowerMaster switches off the modulated signal when $t_{ShutDownWait}$ and $t_{SSO\_Shutdown}$ expire (see Figure 3-21) after ShutDown.Start(Execute). Among other purposes, this time allows shutdown of audio output without audible side effects. If the modulated signal was not switched off within $t_{SlaveShutdown}$, a Slave device may switch off the modulated signal.

If an FBlock desires to continue to communicate, it must notify the PowerMaster after ShutDown.Start(Query) with ShutDown.Result(Suspend).
If the PowerMaster receives ShutDown.Result(Suspend) within time $t_{WaitSuspend}$, it postpones its attempt to switch off for time $t_{RetryShutDown}$, before retrying to shut down. The PowerMaster may override suspend requests from its Slaves and complete the shutdown, for example, to prevent the occurrence of critical conditions like low voltage.

If during $t_{WaitSuspend}$ or $t_{RetryShutDown}$ the shutdown reason ceases to exist, the PowerMaster cancels the shutdown: it does not send out further Shutdown.Start (Query) messages.

For switching off, the PowerMaster calls the respective routine in the Network Service. The status “modulated signal off” travels around the ring in the same way as “modulated signal on” when waking the network. A certain delay time $t_{PwrSwitchOffDelay}$, after switching off the modulated signal, the nodes change to sleep mode.

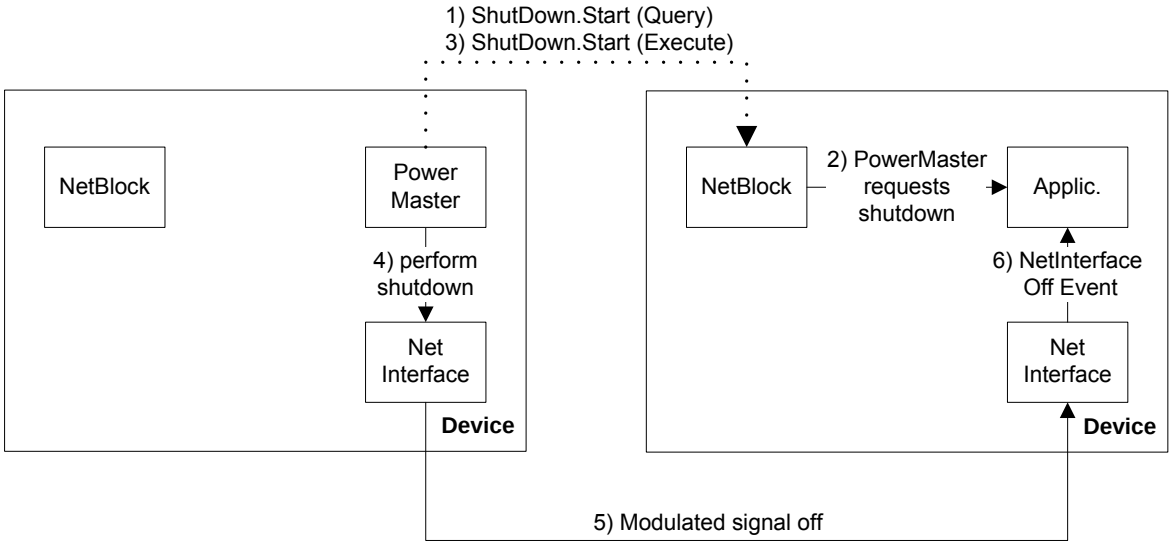

Figure 3-10: Switching off MOST network by starting method ShutDown in every NetBlock

If a device desires to wake the network directly after a shutdown, it has to wait at minimum for $t_{Restart}$ (running from modulated signal off), before it switches on the modulated signal again.

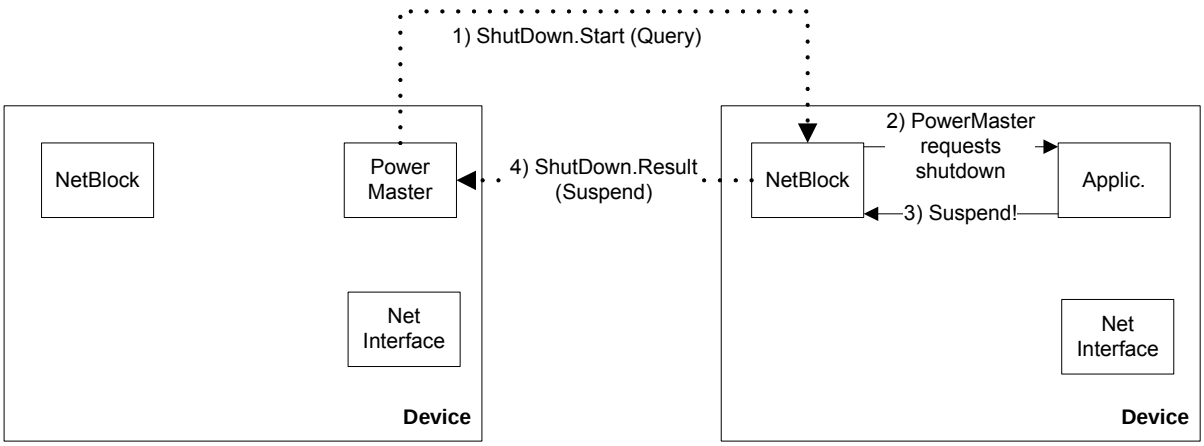

Figure 3-11: Prevention of switching off MOST network via ShutDown.Result (Suspend)

If the modulated signal is switched off during shutdown, for example, by low voltage, Critical Unlock, or fatal error, the PowerMaster must not wake the network in order to finish its shutdown procedure. The PowerMaster must regard the shutdown procedure as complete.

3.1.2.3.3 Device Shutdown

In order to minimize power consumption on system and device level, it is possible to shut down specific MOST devices. This process is called Device Shutdown. Shutting down a device may affect the application on a system level; avoiding such effects is handled by other mechanisms. It is optional to support Device Shutdown.

When a device is shut down, all applications in the device may be shut down with the exception of NetBlock, which is still active with limited functionality: the NetBlock properties FBlockIDs, NodePositionAddress, NodeAddress, GroupAddress, and ShutDown have to be supported.

Also, the device has to know the current System State when it wakes from Device Shutdown, therefore, the NetworkMaster Shadow has to keep track of the current System State even while the device is in Device Shutdown state.

The NetBlock.ShutDown.Start message is used to bring a device into or out of Device Shutdown.

When managing Device Shutdown, this message may be sent to one device or a group of devices, as opposed to Network Shutdown where this message has to be broadcast. The behavior of a device during Network Shutdown is not affected by whether the device is in Device Shutdown or not. Refer to section 3.1.2.3.2 for information about Network Shutdown.

Note: *In Device Shutdown state, the shutdown reason (see 3.1.4.2 Detection of Sudden Signal Off and Critical Unlock) cannot be stored.*

3.1.2.3.3.1 Performing Device Shutdown

The process of shutting down a device or a group of devices can be divided into two stages, a request stage and an execution stage. The request stage is optional.

Request Stage (Optional)

This stage guarantees that a device is not shut down while its FBlocks are communicating with FBlocks on other devices. It is also useful if the PowerMaster wants to shut down a group of devices but only if the whole group is ready.

1. The PowerMaster sends ShutDown.Start (Query) to a single device or a group of devices.
2. The PowerMaster will wait for $t_{WaitSuspend}$ to allow devices to suspend its own shutdown.
3. A device that requires communication will respond with ShutDown.Result (Suspend).
4. If a device responds to the Query, the PowerMaster will wait for $t_{RetryShutDown}$ before trying again.
5. Steps 1 through 4 may be repeated. If no request to suspend the Device Shutdown process is received, $t_{WaitSuspend}$ expires and the execution stage is entered.

Devices with system functions (e.g., NetworkMaster and Connection Manager) shall reject requests by the PowerMaster to perform a Device Shutdown. Those devices return error code 0x07 (“parameter not available”).

Execution Stage

To execute Device Shutdown, the PowerMaster starts method Shutdown with parameter DeviceShutdown in a single device or in a group of devices.

1. The PowerMaster sends ShutDown.Start(DeviceShutdown).
2. A source device de-allocates the bandwidth. A sink device secures any streaming outputs. For sinks, the Mute property remains unchanged.
3. The device unregisters itself from Notification Matrices in other devices, if any.
4. The device unregisters its FBlocks by sending an FBlockIDs.Status with an empty FBlockIDList.
5. The NetworkMaster, using the blocking broadcast address, broadcasts the device's invalid FBlocks.
6. The device can shut down its application but the NetBlock has to stay active.

3.1.2.3.3.2 Waking from Device Shutdown

The device can be woken by the PowerMaster or by the device itself.

Wake-Up by PowerMaster

1. The PowerMaster sends ShutDown.Start(WakeFromDeviceShutdown).
2. The device wakes its application.
3. The device registers its own FBlocks using FBlockIDs.Status(FBlockIDList).
4. When NetworkMaster reports the new FBlocks, they can be used.

Internal Wake-Up

1. The device wakes its application.
2. When the System State is OK or when explicitly asked by the NetworkMaster, the device registers its own FBlocks using FBlockIDs.Status(FBlockIDList).
3. When NetworkMaster reports the new FBlocks, they can be used.

Furthermore, a device that is woken from Device Shutdown must perform the complete "SystemCommunicationInit" procedure (see section *3.1.3.1.1.2 Initialization on Application Level*).

3.1.2.3.3.3 Persistence of Device Shutdown

The state of being in Device Shutdown is not memorized after a system restart.

If the PowerMaster application in a system that supports Device Shutdown is reset in System State OK, the PowerMaster shall broadcast ShutDown.Start(WakeFromDeviceShutdown) at the end of its reset procedure to establish a defined Device Shutdown state.

3.1.2.3.3.4 Response when Device Shutdown is Unsupported

Since Device Shutdown is optional, the NetBlock of a device that does not have support for Device Shutdown responds to a request for Device Shutdown with ErrorCode 0x07 (parameter not available).

3.1.3 Network Management

Network Management is the process by which the NetworkMaster ensures secure communication between applications over the MOST Network. This section describes the conditions that must be met by the NetworkMaster and the NetworkSlaves to enable safe Network Management. The tools used for this process include the control of the System State and the administration of the Central Registry, as well as the Decentral Registries.

Section 3.1.3.1 contains general descriptions of Network Management. Detailed requirements of the behavior of MOST devices regarding Network Management are described in sections 3.1.3.2 through 3.1.3.4. For more implementation specific information and examples refer to Appendix A.

3.1.3.1 General Description of Network Management

3.1.3.1.1 System Startup

This section describes the System Startup following the Init Ready event.

3.1.3.1.1.1 Initialization of the Network

The NetworkMaster is responsible for initializing the network at System Startup. It collects the system configuration by requesting the configuration of each individual NetworkSlave; this is referred to as a System Scan. The collected information is entered into the Central Registry.

The NetworkMaster sets the System State to OK to indicate that the Central Registry is valid or NotOK to indicate that the Central Registry is invalid. When the System State is OK, MOST devices may communicate freely. When the System State is NotOK, communication is limited.

Setting the System State to NotOK resets the system from a network point of view, that is, any network related information is reset in all NetworkSlaves.

The NetworkMaster will set the System State to NotOK whenever an error is caused by a NetworkSlave registration. The NetworkMaster will perform a System Scan as described in section 3.1.3.3.4.

A transition to System State OK indicates the completion of the network initialization.

3.1.3.1.1.2 Initialization on Application Level

After the NetworkMaster has set the System State to OK, initializations that have to do with the interacting of multiple devices on the Application Layer should be performed. However, initialization of the individual applications may start earlier. The application may now initialize communication controlled by itself. This initialization phase is referred to as “SystemCommunicationInit”.

During SystemCommunicationInit, for example, notification is established, so the application of a device may register in the Notification Matrices of those FBlocks from which it desires to get status information.

The system must be prepared for devices connecting to or disconnecting from the network (Network Change Event) and FBlocks being activated and deactivated during runtime. In these cases, the system must run consistently without disturbances and reinitializing phases must be as short as possible. On a Network Change Event, parts of SystemCommunicationInit must be run again, but initialization must not be run completely due to the time this would take.

3.1.3.1.2 General Operation

3.1.3.1.2.1 Finding Communication Partners

When an application seeks a communication partner, that is, an FBlock, it requests the whereabouts of the FBlock from the NetworkMaster. The requesting application receives the available InstIDs of the sought FBlock and the logical node addresses of the devices in which they reside. Alternatively, if a specific FBlock is sought, the InstID may also be specified.

A Controller device may store the information concerning its communication partners in a Decentral Registry. The benefit of having a Decentral Registry is that the NetworkSlave does not have to request the logical node address of its communication partners every time it needs to communicate. The Decentral Registry must be deleted whenever the NetworkMaster sets the System State to NotOK.

3.1.3.1.2.2 Network Monitoring

The NetworkMaster monitors the system for changes and errors. When a Network Change Event is detected, the NetworkMaster must find out if a device has entered or left the network. It scans the network and reports any new information to all NetworkSlaves in the system. This way a device will be notified if one of its communication partners is missing or if new potential communication partners enter the system.

The NetworkMaster may scan the system at any time.

3.1.3.1.2.3 Dynamic FBlock Registrations

It may happen that devices activate and deactivate FBlocks at any time; these changes have to be reported to the NetworkMaster. The NetworkMaster then updates the Central Registry and informs all NetworkSlaves.

3.1.3.2 System States

A MOST Network is in either of two System States, OK or NotOK. The System State reflects the validity of the Central Registry. The NetworkMaster builds and maintains the Central Registry, as well as distributes the System State to all NetworkSlaves.

The NetworkMaster builds the Central Registry by collecting logical node addresses and FBlock configuration from all NetworkSlaves. The system relies on a valid Central Registry, not only because it contains the information used by Controller devices to find their communication partners, but also because it is crucial that a device is informed if one of its communication partners disappears.

The NetworkMaster distributes the System State of the network to the NetworkSlaves by broadcasting Configuration.Status messages, using the blocking broadcast address. The state diagram in Figure 3-12 shows the System States and which events affect the states.

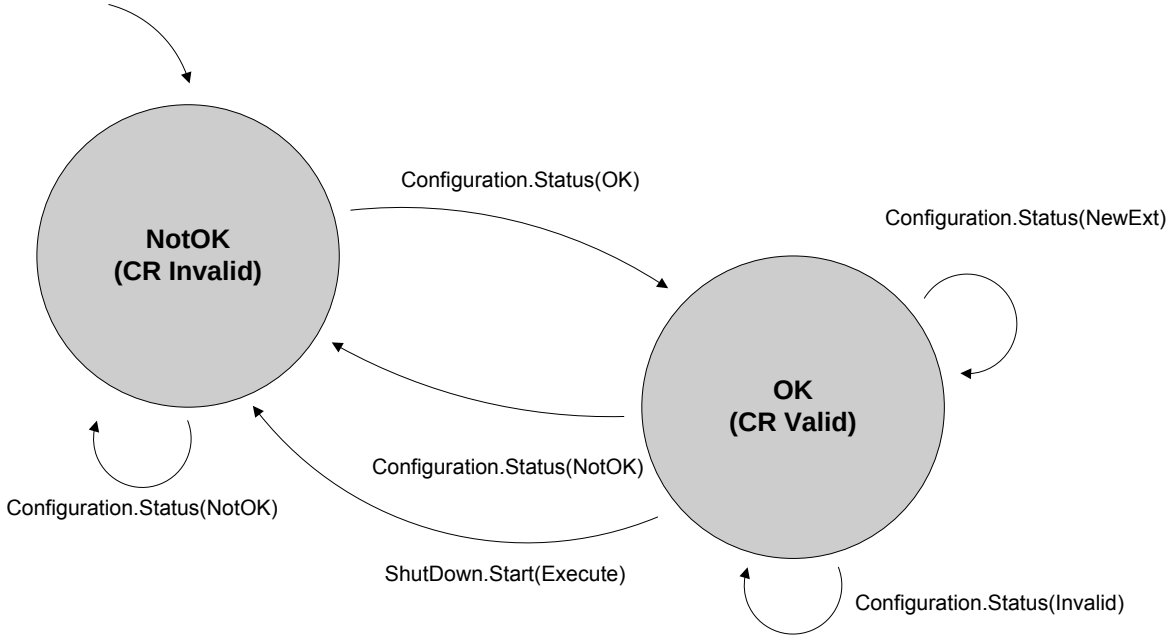

Figure 3-12: States of the network and the Central Registry in NetInterface Normal Operation [1]

The network should be considered to be in a reset state directly following the broadcast of Configuration.Status(NotOK) by the NetworkMaster. Following this event, all devices delete any network configuration related information they may have (e.g., logical node address, Central Registry, Decentral Registry).

Sections 3.1.3.3 NetworkMaster and 3.1.3.4 NetworkSlave describe device specific behavior in the different System States, as well as making transitions between states.

[1] Shutdown.Start(Execute) will not cause the re-calculation of logical node addresses (see 3.1.3.2.1).

3.1.3.2.1 System State NotOK

System State NotOK is always entered after an Init Ready event. In this state, communication must not take place except for special applications that do not rely on a valid registry, in particular the System Scan performed by the NetworkMaster and other optional features that may be done on a per-device, position-dependent basis. The system can fall back into System State NotOK at any time by declaration of the NetworkMaster.

Also, the system is regarded as being in System State NotOK after NetBlock.ShutDown.Start(Execute) has been broadcast.

An optional delay may be specified on a per-system basis between the broadcast of this message and the point in time where the change of state becomes effective. Dynamic logical node addresses are not re-calculated upon this implicit change of state; this is only done when the message Configuration.Status(NotOK) has been received.

Event Configuration.Status	Transition to	Cause	Effect
NotOK	No transition	- Un-initialized NodeAddress in NetworkMaster - Erroneous registration by NetworkSlave.	Network Configuration Reset: - Clear Central Registry - Clear Decentral Registries - Recalculate NodeAddress
OK	System State OK	- Central Registry verified	- Network configuration available in Central Registry - Set up Decentral Registries, where necessary. - (Re-) initialize applications.

Table 3-8: Events in System State NotOK (refer to Figure 3-12)

3.1.3.2.2 System State OK

While in System State OK, the Central Registry is valid. So the exact set of FBlocks in the system, each with its attributes InstID and DeviceID, is defined. Therefore, application communication (i.e., messages with FBlockIDs other than NetBlock or NetworkMaster are allowed) may take place on the Control Channel and Packet Data Channel. All the dynamic communication on the application level within the distributed system should be done only in System State OK.

Event Configuration.Status	Transition to	Cause	Effect
NotOK	System State NotOK	- Erroneous registration by NetworkSlave.	Network Configuration Reset: - Clear Central Registry - Clear Decentral Registries - Recalculate NodeAddress
NewExt	No transition	- New FBlocks are available	- Notify application
Invalid	No transition	- FBlocks were removed	- Notify application

Table 3-9: Events in System State OK (refer to Figure 3-12)

As a result of a Configuration.Status(NotOK) event in System State OK, the following tasks have to be performed by all devices as part of the Network Configuration Reset:

- Every FBlock containing streaming sinks: set the Mute property to “On” for all sinks and disconnect them.
- Every FBlock containing streaming sources: De-allocate all sources.
- Close MOST High Protocol connections.
- Clear any Decentral Registry; the NetworkMaster clears the Central Registry.

- The Connection Manager must delete its connection table.
- Empty Notification Matrix.
- Destroy all ArrayWindows on LongArrays.
- Derive and set the new logical node address (section 3.2.2).

3.1.3.3 NetworkMaster

The device that contains the NetworkMaster FBlock is referred to as the NetworkMaster. There must be one, and only one, NetworkMaster in a MOST Network.

The NetworkMaster controls the System State and administrates the Central Registry. The NetworkMaster monitors the network for certain events and continuously manages incoming information from NetworkSlaves about their current FBlock configuration and whenever necessary informs all NetworkSlaves about updates to the Central Registry.

3.1.3.3.1 Setting the System State

The NetworkMaster distributes the System State by broadcasting Configuration.Status messages, using the blocking broadcast address. More information about the different System States and Configuration.Status messages is available in section 3.1.3.2.

It is critical that Configuration.Status broadcast messages are received by all devices in the MOST network. If at least one MOST NetworkSlave has not received a Configuration.Status broadcast message, the NetworkMaster device performs low-level retries. If these are not successful, the NetworkMaster application should perform additional retries. The number of retries is System Integrator specific.

3.1.3.3.1.1 Setting the System State to OK

By setting the System State to OK, the NetworkMaster confirms the validity of the Central Registry. Therefore, before setting the System State to OK, the NetworkMaster must make sure that all functional addresses are unique in the system (section 2.2.3.3).

The NetworkMaster must do the following when setting the System State to OK:

1. Broadcast Configuration.Status(OK).
2. Trigger initialization of applications in own device.
3. Continue to maintain the Central Registry.

3.1.3.3.1.2 Setting the System State to NotOK (Network Reset)

By broadcasting Configuration.Status(NotOK), the NetworkMaster resets the system (from a network point of view).

The NetworkMaster then executes the actions that are described in section 3.1.3.2.2. The NetworkMaster waits a time $t_{WaitBeforeScan}$ after broadcasting Configuration.Status(NotOK) and afterwards perform a System Scan (section 3.1.3.3.4).

In case the Central Registry becomes unavailable due to an internal failure of the NetworkMaster, the system must be notified with an Configuration.Status(NotOK) message by the device that contains the NetworkMaster.

3.1.3.3.2 Central Registry

The NetworkMaster generates the Central Registry during the initialization of the network and it continues to administrate it until Network Shutdown (section 3.1.2.3.2). The Central Registry is an image of the physical and logical system configuration. It contains the logical node address and the respective FBlocks of each device:

RxTxLog	RxTxPos	FBlockID	InstID
0x0100	0	AudioDiskPlayer	1
		NetworkMaster	10
		ConnectionMaster	1
0x0101	1	AudioDiskPlayer	2
0x0102	2	AM/FMTuner	1
		AudioTapeRecorder	1
0x0103	3	AudioAmplifier	2
Etc.			
MaxNode	MaxNode	HumanMachineInterface	1

Table 3-10: Example of a Central Registry

3.1.3.3.2.1 Purpose

The Central Registry is used when the NetworkMaster checks the system configuration and when devices are searching for communication partners.

3.1.3.3.2.2 Contents

The Central Registry must contain the logical node address and the respective functional addresses (combination of FBlocks and InstIDs) of the FBlocks in each responding MOST device. This information must be made available to all NetworkSlaves.

3.1.3.3.2.3 Responsibility

Any new information gained regarding the system configuration must be entered into the Central Registry and distributed to all NetworkSlaves as described in section 3.1.3.3.6.

The NetworkMaster must only start supervising and store errors for those NetworkSlaves that have answered requests and which are registered in the Central Registry.

3.1.3.3.2.4 Responding to Requests for Information from the Central Registry

The NetworkMaster must respond to requests for CentralRegistry.Get from the NetworkSlaves while the System State is OK. This is described in section 3.1.3.4.4.

If the NetworkMaster receives CentralRegistry.Get from a NetworkSlave while the system is in System State NotOK, the NetworkMaster answers the request with the ErrorCode 0x41 and broadcasts Configuration.Status(NotOK). The message is sent to the blocking broadcast address.

Broadcasting the System State again ensures that every device in the system is updated on the current System State, including and in particular the NetworkSlave that attempted to query the Central Registry.

3.1.3.3.3 Specific Behavior during System Startup

After the Init Ready event, the NetworkMaster must initialize the system. This process depends on the availability of a valid[1] logical node address.

3.1.3.3.3.1 Valid Logical Node Address Not Available

If the NetworkMaster does not have a valid logical node address available at System Startup, it assumes that the entire system must be re-initialized. The NetworkMaster must set the System State to NotOK (section 3.1.3.3.1.2) and then start a System Scan (section 3.1.3.3.4).

3.1.3.3.3.2 Valid Logical Node Address Available

If the NetworkMaster has a valid logical node address at System Startup, it must restore its logical node address and then start a System Scan (section 3.1.3.3.4).
The NetworkMaster must wait a time $t_{WaitBeforeScan}$ before scanning the system for the first time. This latency time allows the system to stabilize after the Init Ready event. This latency time must not exceed $t_{WaitAfterNCE}$.

3.1.3.3.4 Scanning the System (System Scan)

The NetworkMaster scans the system at System Startup and after a Network Change Event (NCE). It may also be instructed to scan the system at any other time.

The NetworkMaster scans the system by requesting the FBlock configuration of each device. The responses from the NetworkSlaves are interpreted as described in section 3.1.3.3.5. Any information gained concerning the configuration of the network must be written to the Central Registry and reported to all NetworkSlaves as described in section 3.1.3.3.6.

The NetworkMaster will set the System State to NotOK whenever an error is caused by a NetworkSlave registration.

3.1.3.3.4.1 Configuration Request Description

During a network scan, the NetworkMaster requests NetBlock.FBlockIDs.Get from each NetworkSlave.

3.1.3.3.4.2 Addressing

The NetworkMaster scans the system by node position addressing. The logical node address of the requested NetworkSlave is contained in the response message.

3.1.3.3.4.3 Non Responding NetworkSlaves

The NetworkMaster must wait until the expiration of $t_{WaitForAnswer}$ for a reply from a NetworkSlave. The NetworkMaster must send another request to the NetworkSlave as described in section 3.1.3.3.4.4.

[1] A valid logical node address is any address within the dynamic or static address ranges as defined in section 3.2.2.1.

3.1.3.3.4.4 Retries of Non Responding NetworkSlaves

When a NetworkSlave does not respond to a request, the NetworkMaster must try again after $t_{DelayCfgRequest1}$ or $t_{DelayCfgRequest2}$. $t_{DelayCfgRequest1}$ is used for the first 20 request attempts after entering NetInterface Normal Operation, after that $t_{DelayCfgRequest2}$ is used.

Refer to section 3.2.9 for more information about timers.

3.1.3.3.4.5 NetworkSlave Continuous cause for System State NotOK

If a node caused a Configuration.Status(NotOK) the third time in direct succession, the NetworkMaster will check if its own address is involved into that conflict. All conflicting NetworkSlaves are ignored until the next NCE or shutdown; the NetworkMaster node must never be ignored.

Ignore in this context means

a) On subsequent system scans FBlockIDs are not requested from the respective nodes anymore.

b) FBlockIDs.Status events from the respective nodes are not evaluated. Other messages from ignored nodes may still be processed.

Note: *The nodes to be ignored are identified by the InstIDs of their NetBlocks (representing the NodePosition).*

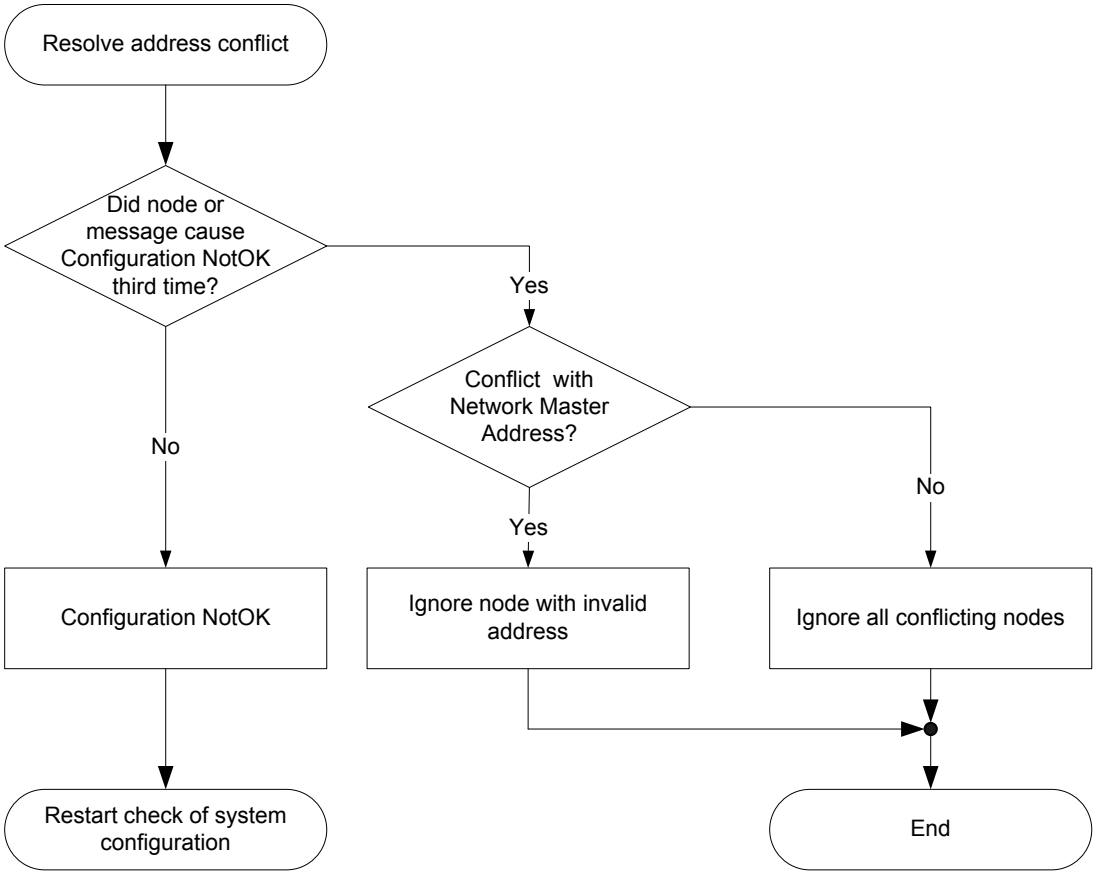

Figure 3-13: NetworkMaster behavior in case of an address conflict

3.1.3.3.4.6 Duration of System Scanning

The NetworkMaster must continue to scan the system until all NetworkSlaves have answered the requests. Refer also to section 3.1.3.3.4.4.

3.1.3.3.4.7 Reporting the Results of a System Scan without Errors

The NetworkMaster must report the result of the System Scan if it has any new information to distribute, such as a change in System State or changes in the FBlock configuration of one or more NetworkSlaves. Refer to sections 3.1.3.3.1 and 3.1.3.3.6.

3.1.3.3.5 Invalid Registration Descriptions

The NetworkMaster interprets the incoming registrations and determines if the registration is accepted. The following are considered to be invalid registrations, but all are not considered erroneous since some may be corrected by the NetworkMaster.

3.1.3.3.5.1 Un-initialized Logical Node Address

If any NetworkSlave, at any time, registers an un-initialized logical node address (section 3.2.2.1), the NetworkMaster must set the System State to NotOK (section 3.1.3.3.1.2), interrupting any ongoing System Scan.

3.1.3.3.5.2 Invalid Logical Node Address

When the NetworkMaster receives a registration from a NetworkSlave in which its logical node address is outside of the specified address range, the NetworkMaster must set the System State to NotOK (section 3.1.3.3.1.2), interrupting any ongoing System Scan.

Refer to section 3.2.2 for more information about the valid address range.

3.1.3.3.5.3 Duplicate Logical Node Addresses

When the NetworkMaster receives a registration from a NetworkSlave in which its logical node address has already been registered by another NetworkSlave, the NetworkMaster must set the System State to NotOK (section 3.1.3.3.1.2), interrupting any ongoing System Scan.

3.1.3.3.5.4 Duplicate InstID Registrations

The NetworkMaster is responsible for the uniqueness of functional addresses (combination of FBlockIDs and InstID) within the entire system. The NetworkMaster must try to resolve the issue of two or more NetworkSlaves registering identical functional addresses.

In the case of duplicate InstIDs, the InstID of the FBlock that is already registered in the Central Registry must not be changed.

The NetworkMaster decides a new InstID for the last FBlock to be registered. It then sets the new InstID in the corresponding NetworkSlave by using the logical node address. If the new InstID was accepted by the NetworkSlave, the NetworkMaster enters the new value into the Central Registry. The NetworkMaster must inform all NetworkSlaves as described in section 3.1.3.3.6 or by ultimately setting the System State to OK.

If the NetworkSlave cannot accept the new InstID, it has to respond with an unchanged NetBlock.FBlockIDs.Status list. The NetworkMaster tries to repeat the procedure with the same or

another InstID value. The maximum number of attempts per instance is to be defined by the System Integrator.

If the request to change the InstID of a conflicting FBlock is not successful, the NetworkMaster must either exclude the FBlock or the entire device from the Central Registry. Figure 3-14 describes the behavior of the NetworkMaster in detail.

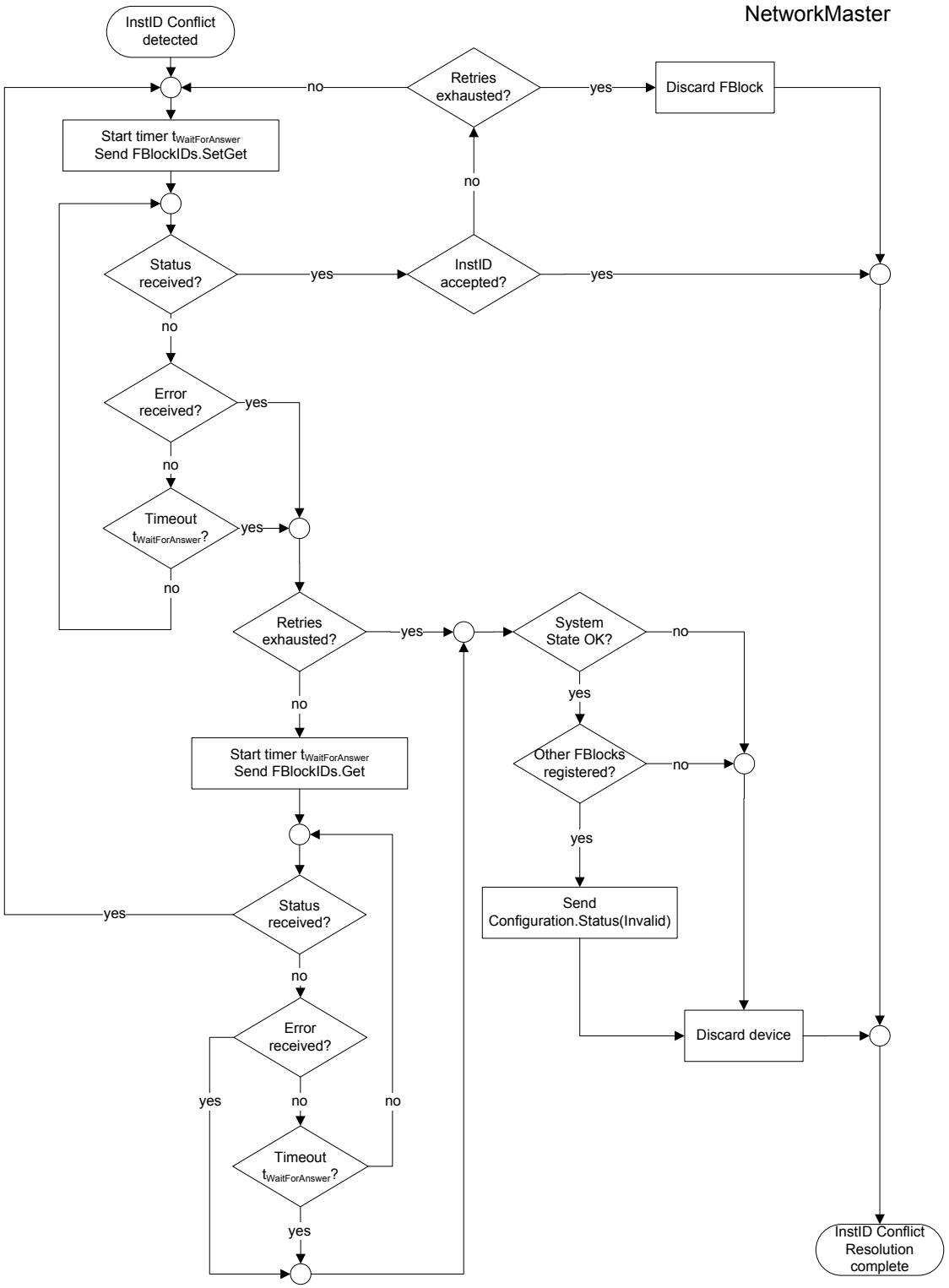

Figure 3-14: InstID conflict resolution

3.1.3.3.5.5 Error Response

A NetworkSlave that answers a request from the NetworkMaster with an error must be treated as a non-responding NetworkSlave (section 3.1.3.3.4.4).

3.1.3.3.6 Updates to the Central Registry

The NetworkMaster must inform all NetworkSlaves about changes of the system configuration. This information may become available during a System Scan or as NetworkSlaves make additional registrations, which are not requested by the NetworkMaster.

This section describes how the NetworkMaster handles changes to the system configuration while in System State OK.

3.1.3.3.6.1 Disappearing FBlocks in System State OK

If the NetworkMaster receives a registration from a NetworkSlave in which there is one or more FBlocks missing compared to the last registration from the same NetworkSlave, the NetworkMaster must update the Central Registry and inform all NetworkSlaves about the missing FBlocks.

This is done by broadcasting (as blocking broadcast):

```
Configuration.Status(Invalid, DeltaFBlockIDList)
```

The DeltaFBlockIDList parameter is a list of the previously registered but now invalid FBlocks. The DeltaFBlockIDList must not contain more entries than a single transfer can transport. If there are more FBlocks, several single transfers must be performed.

If the FBlock 0x03 (ConnectionMaster) is unregistered in the Central Registry, a NetworkMaster.Configuration.Status(NotOK) must be broadcast. Afterwards, a System Scan is performed (see section 3.1.3.3.1.2).

When one or more FBlocks have disappeared the NetworkMaster should inform all NetworkSlaves about the missing FBlocks as quickly as possible, even if this information is gained while scanning the network.

“Own configuration invalid” handling (optional)

If the NetworkMaster receives an OwnConfigInvalid.StartResultAck(…, State=active, DeviceID) from a NetworkSlave, the NetworkMaster starts the “own configuration invalid” handling.
The end of the “own configuration invalid” state is reached when the NetworkMaster has completed the handling of “own configuration invalid” by removing all FBlocks of the device from the Central Registry, in which case it will send OwnConfigInvalid.ResultAck(State=finished) to the NetworkSlave that initiated the handling. The end of the “own configuration invalid” state is also reached when a transition to state Netinterface Off occurs or the System State is set to NotOK.

Note: *The NetworkMaster has to be able to handle OwnConfigInvalid messages from multiple devices.*

3.1.3.3.6.2 Appearing FBlocks in System State OK

If the NetworkMaster receives a registration from a NetworkSlave in which there is one or more additional FBlocks compared to the last registration from the same NetworkSlave, the NetworkMaster must update the Central Registry and inform all NetworkSlaves about the new FBlock.

This is done by broadcasting (as blocking broadcast):

```
Configuration.Status(NewExt, DeltaFBlockIDList)
```

The DeltaFBlockIDList parameter is a list of the new FBlocks, InstIDs, and the corresponding DeviceIDs. The DeltaFBlockIDList must not contain more entries than a single transfer can transport. If there are more FBlocks, several single transfers must be performed.

If this information is gained while scanning the network, the NetworkMaster may continue to scan the system before it informs all NetworkSlaves.

If a device that has already been present in the MOST ring as a node tries to register some more FBlocks and the NetworkMaster cannot include the new FBlocks in the Central Registry because the registry is “full”, there will be no broadcast.

In the case that some but not all of the new FBlocks can be included in the Central Registry, the broadcast list would not be empty but would contain the subset of the instances that were included in the Central Registry:

```
Configuration.Status(NewExt, <partial list>)
```

3.1.3.3.6.3 System scan without any change in Central Registry

The NetworkMaster must broadcast (using the blocking broadcast address) Configuration.Status(NewExt) with an empty list when a network scan that was triggered by an NCE did not detect any changes to the registry.

Note: *Configuration.Status(NewExt) with an empty list can only be sent after the scan has been completed.*

3.1.3.3.6.4 Non-responding Devices in System State OK

If a NetworkSlave, which has registered in the Central Registry since startup, does not respond to a request before $t_{WaitForAnswer}$ expires, the NetworkMaster removes the NetworkSlave from the Central Registry and informs all NetworkSlaves as described in section 3.1.3.3.6.1.

3.1.3.3.7 Miscellaneous NetworkMaster Requirements

3.1.3.3.7.1 Network Change Event (NCE)

When an NCE is detected, the NetworkMaster must start a complete System Scan (section 3.1.3.3.4) after $t_{WaitAfterNCE}$. This also applies to a Verification Scan at System Startup (section 3.1.3.3.3). Any scan in progress when the NCE is detected must be interrupted and restarted.

3.1.3.3.7.2 Positioning of the FBlock NetworkMaster in the MOST Network

The NetworkMaster has to reside in the same device as the TimingMaster.

3.1.3.3.7.3 System Configuration Status Information

Particularly when dealing with diagnostic mechanisms, it is important to have a reliable way of determining when the full system configuration has been reached. This information is, for example, necessary as the trigger for certain checks—such as the comparison of rated and actual configuration—and the precondition for diagnostic services.

Note: *This mechanism is optional.*

FBlock	Function	OPType	Parameter
NetBlock (0x01)	ImplFBlockIDs (0x012)	Get	-
		Status	ImplFBlockIDs
		Error	ErrorCode, ErrorInfo

Parameter ImplFBlockIDs

Basis datatype	Length	Condition	Description
Stream		-	Content: FBlockID, InstID {FBlockID1, InstID1, FBlockID2, InstID2...}

All Application FBlocks (i.e., not NetBlock, EnhancedTestability) implemented by the device are contained in this list with their FBlockID and InstID. FBlocks that are implemented more than once must be included the corresponding number of times. In case of dynamic InstIDs, the InstIDs are numbered as reported by the device after first power on.
In the case that the node is not fully operable, the NetBlock must return an error message if this function is called (0x41 - function temporarily not available). Otherwise, the NetBlock must return a complete list with the implemented FBlocks.

Note: *If the application is available and ImplFBlockIDs returns an empty list, the device does not implement Application FBlocks; e.g. the device is a Controller only.*

After System State OK is entered, the NetworkMaster shall query the property ImplFBlockIDs of each device that reported a non empty list as FBlockIDs.Status.
If a device reported an empty list as FBlockIDs.Status, it is up to the System Integrator to decide whether the NetworkMaster shall query the property ImplFBlockIDs.

The NetworkMaster shall request ImplFBlockIDs exactly once from every device after System State OK was entered. However, the NetworkMaster shall also query the property ImplFBlockIDs after it has received an FBlockIDs.Status message from a device that has not replied to ImplFBlockIDs.Get at all or has replied with ImplFBlockIDs.Error(0x41) before.

A System Integrator may decide to delay that request according to the specific needs.

By comparing the list of implemented FBlocks with the list of registered FBlocks in the Central Registry, the NetworkMaster can determine the current state of the system configuration and makes this information available via the function SystemAvail.

FBlock	Function	OPType	Parameter
NetworkMaster (0x02)	SystemAvail (0xA10)	Get	-
		Status	DeviceAvail, FBlockAvail
		Error	ErrorCode, ErrorInfo

Parameter DeviceAvail

Basis datatype	Range of values	Code	Description
Enum	0x00...0x02	0x00	Incomplete. At least one node is not fully operable because of one of the following conditions: – All nodes have answered but at least one node has answered with the error 0x41. – The property ImplFBlockIDs is not requested, for example, because the system scan has detected an unsolvable DeviceID conflict or the system scan is not finished.
		0x01	Complete. All nodes are fully operable: All available nodes have reported their implemented FBlocks through the property ImplFBlockIDs.
		0x02	Error. At least one node has not answered to ImplFBlockIDs or has returned an error not equal to 0x41.

Parameter FBlockAvail

Basis datatype	Range of values	Code	Description
Enum	0x00...0x01	0x00	Incomplete: all other cases (Not all FBlocks reported by the property ImplFBlockIDs of the available devices are currently registered in the CentralRegistry or the state of DeviceAvail is still Incomplete.)
		0x01	Complete: DeviceAvail is Complete and all FBlocks reported by ImplFBlockIDs are registered in the CentralRegistry.

3.1.3.4 NetworkSlave

All devices that do not contain the NetworkMaster FBlock are called NetworkSlaves. A NetworkSlave must keep the NetworkMaster informed about its current FBlock configuration.

In particular, the NetworkSlave is responsible for ensuring that FBlockID.Status messages, which register or unregister FBlocks, are successfully transmitted to the node containing the NetworkMaster.

3.1.3.4.1 Decentral Registry

Every device that controls other devices should build a Decentral Registry in which it registers its communication partners. The device that contains the NetworkMaster—and therefore implements the Central Registry—does not necessarily build a Decentral Registry.

A Decentral Registry contains the functional addresses and the respective logical node address:

Functional Address (FBlockID.InstID)	Device Containing the FBlock (Logical Node Address = DeviceID)
AudioAmplifier.1	0x0105
AudioAmplifier.2	0x0103
AM/FMTuner.1	0x0107
AudioDiskPlayer.1	0x0107

Table 3-11: Example of a Decentral Registry

3.1.3.4.1.1 Building a Decentral Registry

The information stored in the Decentral Registry is gained from the Central Registry. The Decentral Registries are only re-built on demand, that is, not directly following a transition to System State OK.

3.1.3.4.1.2 Updating the Decentral Registry

The FBlock entries stored in the Decentral Registry must match the entries of the respective FBlock in the Central Registry. When the NetworkMaster informs of updates to the Central Registry, the Decentral Registry must be updated accordingly for the registered FBlocks.

3.1.3.4.1.3 Deleting the Decentral Registry

The Decentral Registry must be cleared when the device enters or assumes state NotOK.

3.1.3.4.2 Specific Startup Behavior

Following the Init Ready event, the NetworkSlave initializes its logical node address and services requests from the NetworkMaster.

3.1.3.4.2.1 Behavior when a Valid Logical Node Address is not Available at System Startup

If the NetworkSlave does not have valid[1] logical node address available at System Startup, it must set its logical node address to the value of an un-initialized logical node address (section 3.2.2.1) and services requests from the NetworkMaster until the NetworkMaster sets the System State.

3.1.3.4.2.2 Behavior when a Valid Logical Node Address is Available at System Startup

If the NetworkSlave has a valid[1] logical node address stored from the previous system run, it uses that logical node address and services requests from the NetworkMaster until the NetworkMaster sets the System State.

3.1.3.4.2.3 Deriving the Logical Node Address of the NetworkMaster

A NetworkSlave must derive the logical node address of the NetworkMaster from the following events (i.e., from the Source Address of the following messages):

a) NetBlock.FBlockIDs.Get, if the System State is NotOK.

b) NetworkMaster.Configuration.Status(OK, NewExt, or Invalid)

If the address that is derived from the Configuration.Status message differs from the address derived from FBlockIDs.Get, the address that depends on Configuration.Status must be used.

A NetworkSlave resets the NetworkMaster address (to 0xFFFF) on the following events:

a) each transition to state NetInterfacePowerOff.

b) each transition to System State NotOK.

c) reception of NetBlock.ShutDown.Start (Execute).

[1] A valid logical node address is any address within the dynamic or static address ranges as defined in section 3.2.2.1.

3.1.3.4.3 Normal Operation of the NetworkSlave

3.1.3.4.3.1 Behavior in System State OK

A NetworkSlave may communicate freely while the System State is OK.

3.1.3.4.3.2 Behavior in System State NotOK

While the System State is NotOK, a NetworkSlave must not initiate any communication except for special applications that do not rely on a valid Central Registry, such as optional features that can be done on a per-device, position-dependent basis.

A NetworkSlave must not send a NetBlock.FBlockIDs.Status(FBlockIDList) message in System State NotOK without being requested explicitly by the NetworkMaster.

3.1.3.4.3.3 Responding to Configuration Requests by the NetworkMaster

The NetworkSlave responds to requests for FBlock configuration from the NetworkMaster at all times, regardless of the current System State. The response must be sent before the expiration of t_{Answer}.

The NetworkSlave must report all FBlocks that are currently active from a network point of view. The FBlocks 0x00 (Network Service), 0x01 (NetBlock), and 0x0F (Enhanced Testability) are neither listed in NetBlock.FBlockIDs.Status messages nor in the Central Registry. In general, it is not recommended to include FBlocks that have their InstIDs coupled with the current node position in the NetBlock.FBlockIDs.Status list or the Central Registry.

If the NetworkSlave does not have any active FBlocks, it must respond with an empty FBlockIDList.

3.1.3.4.3.4 Reporting Configuration Changes to the NetworkMaster

When the FBlock configuration of a NetworkSlave changes, it must report this change to the NetworkMaster; however, it must not do so if the current System State is NotOK (section 3.1.3.4.3.2).

3.1.3.4.3.5 Failure of an FBlock in a NetworkSlave

This behavior is described in section 3.1.5.4.

3.1.3.4.3.6 Failure of a NetworkSlave Device

This behavior is described in section 3.1.5.4.

3.1.3.4.3.7 Unknown System State

If the NetworkSlave does not know the current System State, it must assume that the System State is NotOK. For determining of System State, refer to section 3.1.3.4.3.8.

Note: *The device also has to assume state NotOK when the NetInterface enters normal operation.*

3.1.3.4.3.8 Determining the System State

The current System State must be determined from the Configuration.Status message, which is broadcast by the NetworkMaster, using the blocking broadcast address. The Configuration.Status (NotOK) implies the system status being NotOK. All other Configuration.Status messages imply the System State is OK, for example,

- Configuration.Status(OK)
- Configuration.Status(NewExt)
- Configuration.Status(Invalid)
- Configuration.Status(NewExt, <empty>)

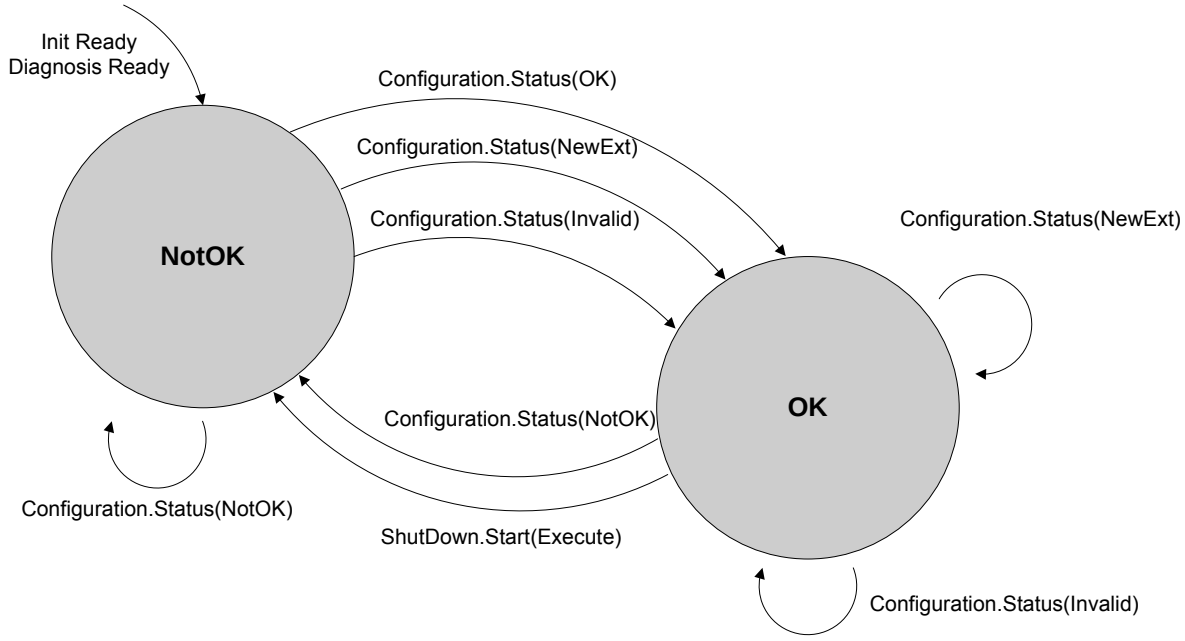

Figure 3-15: NetworkSlave determines the System State in NetInterface Normal Operation

3.1.3.4.3.9 Finding Communication Partners

The Central Registry must be used if the application of a device seeks a logical node address. Note that this must be done only if the information is not already available in a Decentral Registry (section 3.1.3.4.1).

3.1.3.4.3.10 Reaction to Configuration.Status(OK) when in System State NotOK

When the NetworkMaster sets the System State to OK the NetworkSlave:

1. Uses its current Decentral Registry or rebuilds a Decentral Registry when necessary.
2. (Re-) initializes the application.

The System State is set to OK. For additional information, refer to section 3.1.3.2.

3.1.3.4.3.11 Reaction to Configuration.Status(OK) when in System State OK

If a device receives Configuration.Status(OK) in state OK, it ignores the message.

3.1.3.4.3.12 Reaction to Configuration.Status(NotOK) when in System State NotOK

The NetworkMaster sends this message to reset all NetworkSlaves from a network point of view. All NetworkSlaves must:

1. Clear any Decentral Registry.
2. Derive and set the new logical node address (section 3.2.2).

The System State remains in NotOK. For additional information, refer to section 3.1.3.2.

3.1.3.4.3.13 Reaction to Configuration.Status(NotOK) when in System State OK

When the NetworkMaster sets the System State to NotOK, the NetworkSlave must perform the actions that are described in section 3.1.3.2.2; it services requests from the NetworkMaster while waiting for the System State to be set to OK. For additional information, refer to section 3.1.3.2.

3.1.3.4.3.14 Reaction to Configuration.Status(NewExt)

One or more FBlocks have entered the system and the Central Registry has been updated with the FBlocks supplied in the message. These FBlocks should be added to the Decentral Registry, if they are used by the device.

This message is only sent in System State OK. The System State remains in OK. For additional information, refer to section 3.1.3.2.

3.1.3.4.3.15 Reaction to Configuration.Status(Invalid)

One or more FBlocks have left the system and the FBlocks supplied in the message have been removed from the Central Registry. These FBlocks have to be removed from the Decentral Registry if entered.

If a device receives a Configuration.Status(Invalid) with (part of) its own FBlockID(s) and InstID (“own configuration invalid”), it has to reinitialize all applications and send an empty FBlockID list to the NetworkMaster.

The following actions are taken before reinitialization:

- a sink device secures its output and sets the Mute property to “On”
- a source device de-allocates the bandwidth
- packet data connections are closed
- all notifications are cleared

If afterwards the system remains in System State OK, the NetworkSlave registers its FBlocks, which prompts the NetworkMaster to send Configuration.Status(NewExt) to the blocking broadcast address.

“Own configuration invalid” handling (optional)

During System State OK, when the NetworkMaster removes FBlocks of a NetworkSlave device from the Central Registry, it sends Configuration.Status(Invalid) containing the de-registered FBlocks to the NetworkSlave. The NetworkSlave identifies its own FBlocks and as a result the NetworkSlave sends OwnConfigInvalid.StartResultAck(…, State=active, DeviceID) to the NetworkMaster; this starts the “own configuration invalid” handling. The NetworkSlave then sends an FBlockIDs.Status with an empty list.

The NetworkSlave cleans up streaming connections and packet data connections, clears the Notification Matrix, and reinitializes its FBlocks.

The NetworkSlave ignores all own FBlocks in Configuration.Status(Invalid) or Configuration.Status(NewExt) messages from the NetworkMaster until the “own configuration invalid” handling has completed. During the “own configuration invalid” handling, the NetworkSlave answers FBlockIDs.Get requests with an FBlockIDs.Status message that contains an empty list.

The end of the “own configuration invalid” state is signaled either by a transition to state NetInterface Off, Configuration.Status(NotOK), or OwnConfigInvalid.ResultAck(State=finished).

3.1.3.4.4 Seeking Communication Partner

To seek all instances of a specific FBlock, the seeking Controller sends the following command to the NetworkMaster (NWM):

```
Controller -> NWM : NetworkMaster.CentralRegistry.Get ( FBlockID )
```

The NetworkMaster contains the Central Registry, which represents an image of the physical and logical system configuration. It answers with a list of all matching entries of the Central Registry with logical node address and functional address[1]:

```
NWM -> Controller :  NetworkMaster.CentralRegistry.Status (
                     RxTxLog,FBlockID,InstID,
                     RxTxLog,FBlockID,InstID,...)
```

Optionally, the InstID can also be specified to search for a certain FBlock:

```
Controller -> NWM : NetworkMaster.CentralRegistry.Get ( FBlockID,InstID )
```

If the respective FBlockID/InstID combination does not exist, the NetworkMaster replies with ErrorCode 0x07 “Parameter not available”. In case the FBlockID does not exist, in ErrorInfo, it returns 0x01 and the FBlockID. If the FBlockID exists but the requested InstID does not, it returns 0x02 and the InstID.

Note: *Any parameter (FBlockID or InstID) that is not provided when using CentralRegistry.Get, is assigned the wildcard value 0xFF. Thus, for example, CentralRegistry.Get without parameters is identical to CentralRegistry.Get (0xFF, 0xFF).*

[1] The NetworkMaster answers to requests even if the requesting device is not contained in the Central Registry.

The following table describes the response of the NetworkMaster to all FBlockID/InstID combinations.

InstID / FBlockID	0x00	0x01…0xFE	0xFF
0x00	These FBlockID/InstID combinations are not allowed. The NetworkMaster will respond with ErrorCode 0x06 (parameter out of range).		
0x01…0xFE	If InstID 0x00 of the requested FBlock exists, it will be reported. Otherwise, one existing InstID of that FBlock will be reported. However, while the system remains in System State OK and the InstID is still present, the same InstID must be reported in subsequent calls with identical parameters.	If it exists, the specific FBlockID.InstID combination will be returned.	All existing InstIDs of the requested FBlock will be reported.
0xFF	These FBlockID/InstID combinations are not allowed. The NetworkMaster will respond with ErrorCode 0x06 (parameter out of range). FBlockID 0xFF requires InstID 0xFF.		The entire Central Registry is returned.

Table 3-12: FBlockID, InstID combinations for querying the Central Registry

3.1.3.4.5 Requesting FBlock Information from a Device

To obtain information about the FBlocks contained in a device, every NetBlock has the property **FBlockIDs** (0x000). It will be requested in the following way:

```
NetworkMaster -> ??? : NetBlock.FBlockIDs.Get
```

The NetworkSlave answers with a list of the contained FBlockIDs. The FBlock that most characterizes the device (e.g., Tuner in a radio device) is listed first. The NetBlock and FBlock EnhancedTestability are not listed, as they are mandatory FBlocks in every device:

```
??? -> Controller : NetBlock.FBlockIDs.Status (FBlockID1,InstID1,
                                                FBlockID2,InstID2...
                                                FBlockIDN,InstIDN)
```

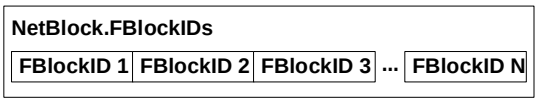

Figure 3-16: Reading the FBlocks of a device from NetBlock

Note: *"Controller" refers to the NetworkMaster in this case. The more generic term "Controller" is used because the NetworkSlave cannot verify in all cases that the sender is the NetworkMaster.*

3.1.3.4.6 Requesting Functions from an FBlock

In an adaptable system it may happen that a Controller does not know exactly which functions are available in an FBlock (e.g., simple or high-end audio amplifier). Therefore, every FBlock that is listed in the Central Registry has the function **FktIDs** (0x000)[1]. It is read as follows:

```
Controller -> ??? : FBlockID.InstID.FktIDs.Get
```

Within an FBlock, FktIDs between 0x000 and 0xFFF (4096 different FktIDs) can be available. The FktIDs are assigned as described in 2.2.1 on page 40. This raises the problem of a compact response, if the functions contained in an FBlock are requested. It is solved by a mechanism derived from the Run Length Encoding (RLE). A bit field is built where the first bit is set to 1 if FktID 0x000 is available; the second bit is set to 1 if FktID 0x001 is available, and so on. Such a bit field may look like:

FktID		000	001	002	003	004	005	006	...	021	022	023	024	...	A00	A01	A02	A03	...	FFF
Bit field	1	1	1	0	0	1	1	0	0	0	1	1	0	0	1	1	0	0	0	0

The answer lists only the positions (FktIDs) where the bit state changes, beginning with an initial bit state of 1.

For the example shown above, the result would be:

```
??? -> Controller : FBlockID.InstID.FktIDs.Status (002 004 006 022
                                                   024 A00 A02 0)
```

The last 0 represents a stuffing nibble.

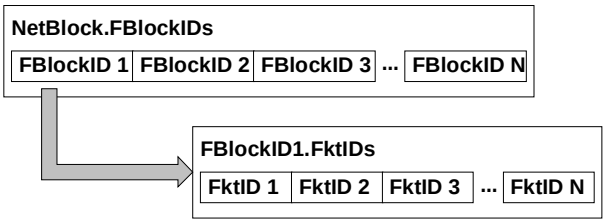

Figure 3-17: Requesting the functions contained in an Application FBlock

3.1.3.4.7 Extended FBlock Identification

To distinguish between the functionality of similar FBlocks with the same FBlockID, the property *FBlockInfo* has to be implemented by each FBlock.

The function FBlockInfo provides information about the FBlock name, the name of the instance, the corresponding MOST Specification version, and the version of the FBlock itself. All this information is provided in text form.

The name of the instance is a string, which is assigned by the System Integrator to allow a further differentiation of the application, for example, “DISP_L” for a left display or “DISP_R” for a right display.

Additionally, FBlockInfo can provide maturity information for each function that is implemented by the FBlock. For those FBlocks that do not appear in the Central Registry, maturity status is usually reported as “Unknown”.

[1] The function FktIDs (0x000) is optional for FBlocks that are not listed in the Central Registry (e.g., FBlockID 0x00, NetBlock, and EnhancedTestability).

3.1.4 Diagnosis

3.1.4.1 Ring Break Diagnosis

3.1.4.1.1 Functional Description

Ring break diagnosis (RBD) can be regarded as a process with 3 phases:

- Phase 1: Activation

  An external trigger is applied to the device to start the ring break diagnosis. The trigger event is system specific (e.g., switch-to-power detection or electrical wake-up line). The timing constraint t_{Diag_Start} specifies the maximum delay between occurrence of the trigger and entering phase 2.

- Phase 2: Diagnosis

  The actual diagnosis takes place during this phase. Each device stores its diagnosis result when this phase is finished.

- Phase 3: Delivery of result

  During this phase the diagnosis result from phase 2 is delivered to a dedicated application. This phase is optional.

Phase 2, the actual diagnosis phase, is divided into three time slots. The relation between these slots and the timers used in the following descriptions is shown in this diagram:

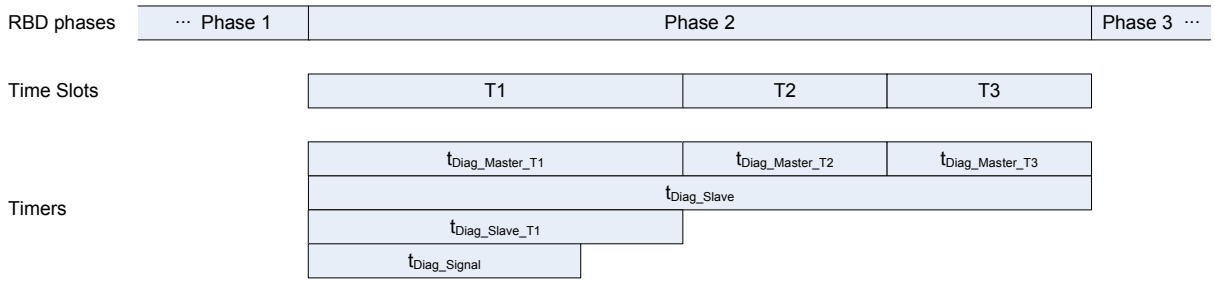

Figure 3-18: Ring break diagnosis phases and time slots

The precondition for a valid ring break diagnosis result is the existence of exactly one designated TimingMaster in the system.

3.1.4.1.1.1 Behavior of a TimingSlave device

If a device sees a signal within t_{Diag_Signal} and Stable Lock within T1, no ring break exists between this and the preceding device. Therefore, it waits for the end of the RBD (t_{Diag_Slave}). If the input signal disappears during this time, the output is switched off for $t_{Restart}$ before it can follow the incoming signal again. If the device sees the System Lock Flag, no ring break was detected and the device switches to NetInterface Normal Operation mode.
If the device sees no signal within t_{Diag_Signal}, it assumes a ring break to be in front of it and switches to TimingMaster mode. This state is left only if a signal is detected or t_{Diag_Slave} is finished. If a signal is detected, another TimingMaster is in the ring and the device switches to TimingSlave mode.

When the device sees a signal, but cannot detect a Stable Lock within T1, it assumes excessive attenuation in the previous network section. Therefore it switches to TimingMaster mode and stays there until t_{Diag_Slave} is finished.

State	Description
RBD_S_Start	In this state, the node is waiting for signal at input or timeout(t_{Diag_Signal}). The state switches to RBD_S_Slave as soon as a signal at input occurs. If timeout(t_{Diag_Signal}), the State switches to RBD_S_NoSig.
RBD_S_Slave	In this state, the node received a signal at input and it is waiting for a Stable Lock or timeout($t_{Diag\_Slave\_T1}$). The state switches to RBD_S_NetOff1, if no signal at input occurs. If timeout($t_{Diag\_Slave\_T1}$) and no StableLockOccurrence, the state switches to RBD_S_Sig. If timeout($t_{Diag\_Slave\_T1}$) and StableLockOccurrence, the state switches to RBD_S_Lock.
RBD_S_Lock	In this state, the node had a Stable Lock and is waiting for timeout(t_{Diag_Slave}) or the System Lock Flag (closed ring). The state switches to RBD_S_NetOff2, if no signal at input or timeout(t_{Diag_Slave}).
RBD_S_Sig	In this state, the node never had a Stable Lock. The state switches to RBD_S_NetOff2, if timeout(t_{Diag_Slave}).
RBD_S_NoSig	In this state, the node never received a signal at input. The state switches to RBD_S_NetOff2, if timeout(t_{Diag_Slave}) or signal at input.
RBD_S_NetOff1	In this state, the node is waiting for signal at input or timeout($t_{Diag\_Slave\_T1}$). If timeout($t_{Diag\_Slave\_T1}$), timeout($t_{Restart}$) and no StableLockOccurrence, the state switches to RBD_S_Sig. If timeout($t_{Restart}$) and signal at input, the state switches to RBD_S_Slave. If timeout($t_{Diag\_Slave\_T1}$), timeout($t_{Restart}$) and StableLockOccurrence, the state switches to RBD_S_NetOff2.
RBD_S_NetOff2	In this state, the node is waiting for the end of RBD or signal at input. If timeout($t_{Restart}$) and signal at input, the state switches to RBD_S_Lock.
RBD_S_NetOff3	If the ring could not be closed because of a ring break, devices shall not be restarted by incoming modulated signal until $t_{Diag\_Restart}$ has passed.

Table 3-13: Ring break diagnosis states - TimingSlave

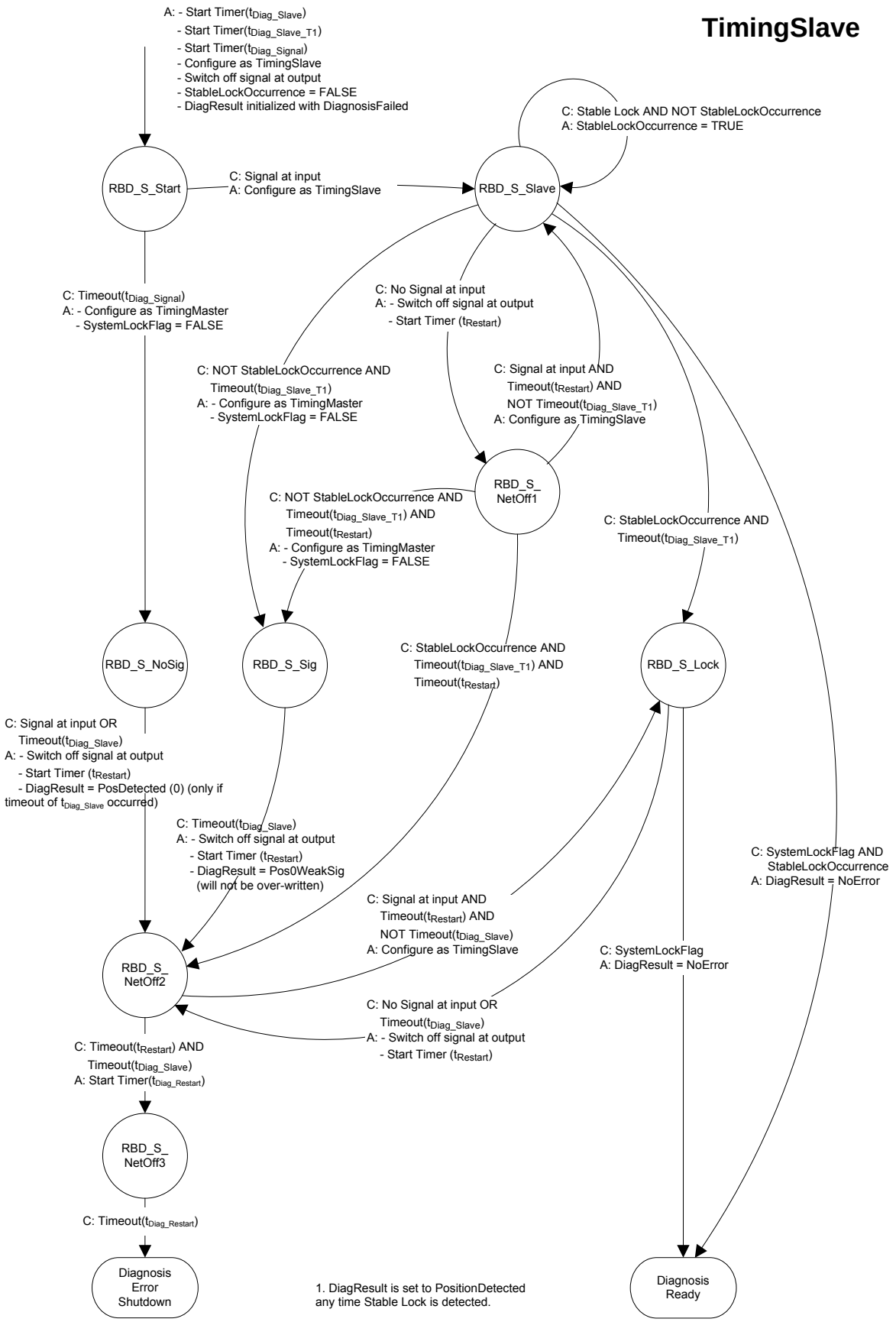

Figure 3-19: Ring break diagnosis – state diagram TimingSlave

3.1.4.1.1.2 Behavior of TimingMaster device

A TimingMaster device outputs a signal instantly when the RBD is started. If it detects a signal and a Stable Lock during T1, it determines whether the ring is closed. If the ring is closed, no ring break is detected, so the device sets the System Lock Flag and switches to NetInterface Normal Operation mode.

If the device does not detect a signal within T1, it assumes a ring break to be in front of it and finishes the RBD.

When the device sees a signal but no Stable Lock at the end of T1, it switches to TimingSlave mode. If it sees Stable Lock within T2, another TimingMaster is active and the device stays in TimingSlave mode during T3. But if it sees no Stable Lock, it assumes excessive attenuation in the previous network section and switches back to TimingMaster mode during T3.

State	Description
RBD_M_Start	In this state, the node is waiting for a Stable Lock or timeout($t_{Diag\_Master\_T1}$). If timeout($t_{Diag\_Master\_T1}$) and signal at input, the state switches to RBD_M_Slave. If timeout($t_{Diag\_Master\_T1}$) and no signal at input, the state switches to RBD_M_NetOff2.
RBD_M_Slave	In this state, the node received a signal at input and it is waiting for a Stable Lock or timeout($t_{Diag\_Master\_T2}$). The state switches to RBD_M_NetOff1, if no signal at input. If timeout($t_{Diag\_Master\_T2}$) and no StableLockOccurrence, the state switches to RBD_M_Sig. If timeout($t_{Diag\_Master\_T2}$) and StableLockOccurrence, the state switches to RBD_M_Lock.
RBD_M_Lock	In this state, the node had a Stable Lock and is waiting for timeout($t_{Diag\_Master\_T3}$) or the System Lock Flag (ring closed). The state switches to RBD_M_NetOff2, if no signal at input or timeout($t_{Diag\_Master\_T3}$).
RBD_M_Sig	In this state, the node never had a Stable Lock. The state switches to RBD_M_NetOff2, if timeout($t_{Diag\_Master\_T3}$).
RBD_M_NetOff1	In this state, the node is waiting for signal at input or timeout($t_{Diag\_Master\_T2}$). If timeout($t_{Diag\_Master\_T2}$), timeout($t_{Restart}$) and no StableLockOccurrence, the state switches to RBD_M_Sig. If timeout($t_{Restart}$) and signal at input, the state switches to RBD_M_Slave. If timeout($t_{Diag\_Master\_T2}$), timeout($t_{Restart}$) and StableLockOccurrence, the state switches to RBD_M_NetOff2.
RBD_M_NetOff2	In this state, the node is waiting for the end of RBD or signal at input. If timeout($t_{Restart}$) and signal at input, the state switches to RBD_M_Lock.
RBD_M_NetOff3	If the ring could not be closed because of a ring break, devices shall not be restarted by incoming modulated signal until $t_{Diag\_Restart}$ has passed.

Table 3-14: Ring break diagnosis states - TimingMaster

TimingMaster

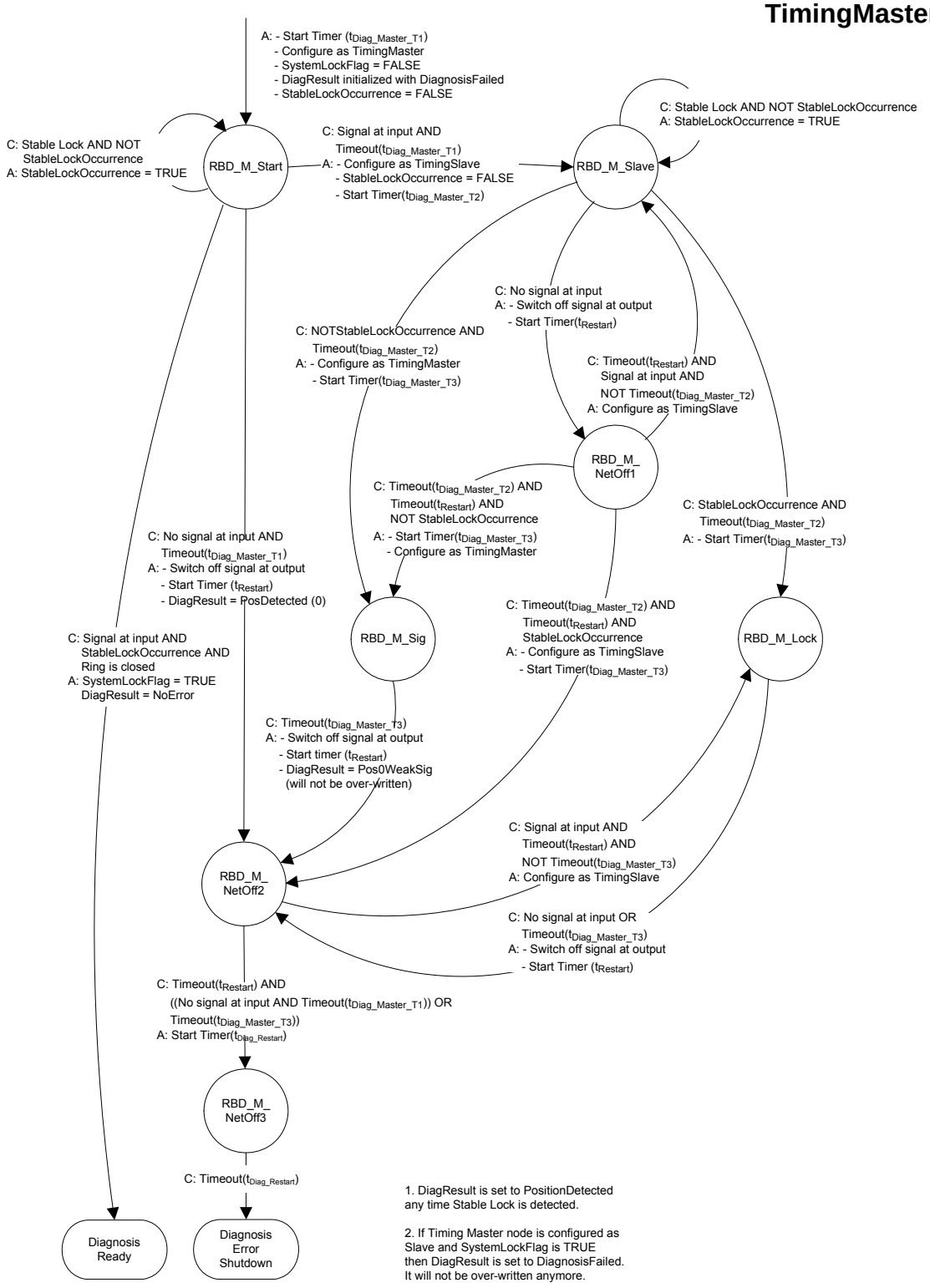

Figure 3-20: Ring break diagnosis – state diagram TimingMaster

3.1.4.1.2 Ring Break Diagnosis Result

Each device stores the relative ring position which is seen during Stable Lock and reports it as diagnosis result when a ring break is detected. The table lists the possible results of ring break diagnosis and the corresponding events.

Diagnosis Result	Diagnosis Info	Description	Event
No error	-	No error detected. Transition to NetInterface Normal Operation.	Diagnosis Ready
Ring Break (Position detected)	Relative position.	Ring break detected. The result indicates the relative position of ring break.	Diagnosis Error Shutdown
Weak Signal (Excessive attenuation at input)	-	Activity is present, but Stable Lock could not be established. Position 0 is assumed because there is no valid relative node position available.	Diagnosis Error Shutdown
Diagnosis failed	-	The ring break diagnosis is inconclusive.	Diagnosis Error Shutdown

Table 3-15: Ring break diagnosis results

When NetInterface state Diagnosis Result is used (see 3.1.2.2 NetInterface), the device in TimingMaster mode (first node downstream the faulty network section) switches on the signal again and broadcasts the message NetBlock.RBDResult.Status() within RBD phase 3. The message contains the status of the test (Activity/NoActivity) and the diagnostic identifier of the device (DiagID).

The content of the DiagID is defined by the System Integrator; its length has to be chosen so that the message can be sent as an unsegmented control message.

3.1.4.2 Detection of Sudden Signal Off and Critical Unlock

When a Sudden Signal Off (SSO) or Critical Unlock (CU) occurs in Normal Operation mode, the device behind the SSO or CU switches to TimingMaster mode without generating a Critical Unlock. Within $t_{ShutDown}$, it sets the Shutdown Flag in the administrative area of the MOST frames, which notifies all other devices about the shutdown. It delays switching off the signal at the output for $t_{SSO\_ShutDown}$ to ensure that the following devices can detect the Shutdown Flag. The detection of the Shutdown Flag has to be secured.

The detecting device stores the cause of the fault (SSO or CU); all other devices have noticed the Shutdown Flag and consequently store “No fault saved”. All causes of faults must be stored by the application so that the information is still available after a reset. The initial value is “No result available”.

The designated TimingMaster cannot detect the Shutdown Flag which was set by the detecting device that temporarily operates in TimingMaster mode. Therefore, the designated TimingMaster will also detect an SSO or a CU fault and store it. This has to be considered when the stored faults are evaluated after the next start of the system.

A TimingSlave device which has detected an SSO or a CU will not close the bypass after switching off the signal. If there is any activity at the input of the device, it must not restart as long as the Shutdown Flag is detected in the incoming signal, in order to prevent that the network is restarted before the shutdown is complete.

Also, the PowerMaster has to set the Shutdown Flag when it performs the normal shutdown procedure.

This detection applies only to NetInterface State Normal Operation. In any other state, the signal is switched off, without propagating the Shutdown Flag, and without saving any shutdown reason.

The following flow chart shows this behavior:

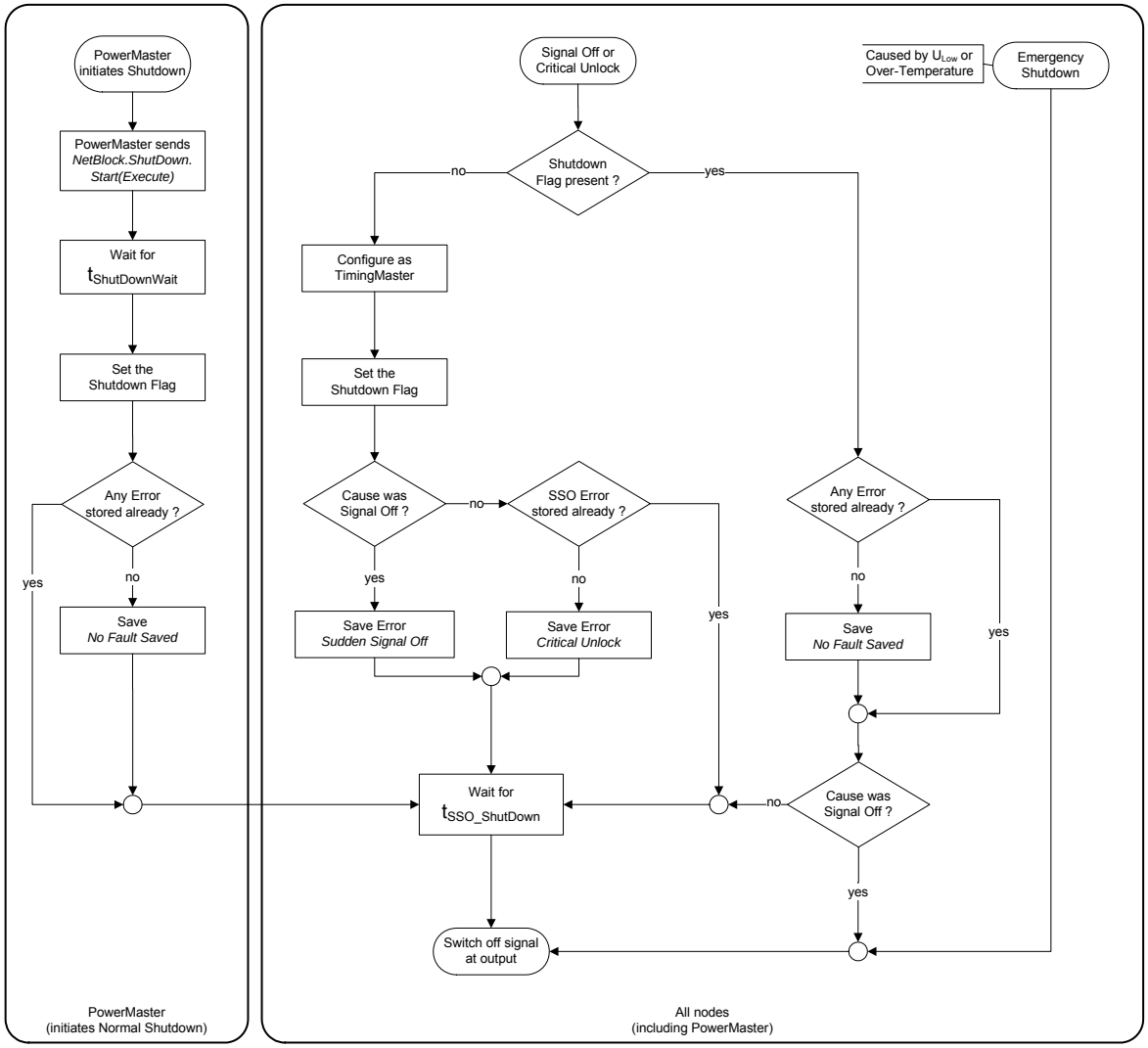

Figure 3-21: Shutdown causes in state NetInterface Normal Operation

3.1.4.3 Shutdown Result Analysis

After the restart of the network, a central component may query all nodes for stored faults. This query takes place when the System Configuration State is OK, i.e., after the NetworkMaster has sent the Configuration.Status(OK) message. This unique Controller sends the following single cast message to all devices:

```
NetBlock.ShutDownReason.Get()
```

The devices will answer with the corresponding Status message or with error “FktID not available” if the application is not available yet. When it has gathered the information from all devices, it sends

```
NetBlock.ShutDownReason.Set(0x00)
```

to all nodes. Only now all devices delete the stored shutdown reason. This ensures that in the case of a reappearing fault the result of the analysis is not corrupted.

In cases where the TimingMaster detects an error, the unique Controller must ignore the shutdown reason in the TimingMaster if another device also reports the fault.

This NetBlock function is mandatory for all devices.

3.1.4.4 Coding Error Counter

For diagnostic purposes the Network Service of each MOST node must provide a function to determine the occurrence of coding errors and to read out this information from an application. Coding errors are violations of the coding scheme.
The coding error counter counts corrupted frames depending on the operating mode. The implementation must prevent a wrap-around of the counter.

Retimed Bypass Mode
If the device is in Retimed Bypass Mode, which is a special test mode used for physical layer testing, the counter counts all corrupted frames independent of the NetInterface state. The Retimed Bypass Mode is activated by the FBlock EnhancedTestability API.

Normal Operation Mode
In Normal Operation mode, coding errors must not be counted during start-up or shutdown, that is, before reaching Configuration.Status(OK) and after ShutDown.Start(Execute).
Additionally, coding errors must not be counted when more serious error events are detected, such as an unlock, a Critical Unlock, or a Sudden Signal Off. The counter is reset on transition to state NetInterface Normal Operation and whenever the coding error counter is read.

3.1.5 Error Management

In the network, the following errors may occur on a high level:

- **Failure of a NetworkSlave Device:** Error that leads to either a reset of the whole device or reset of the failing application.
- **Critical Voltage:** The NetInterface works normally, the device can communicate. On a recovery from this state, the network does not need to be initialized again.
- **Over-Temperature:** Depending on the temperature, four different alert levels are defined.

Additionally, errors may occur on a lower level:

- **Fatal Error:** Error that leads to the interruption of the ring, to the breakdown of the network, or that means the network cannot be initialized (low voltage, ring break, defective Physical Interface unit).
- **Unlock:** The PLL of the MOST Network Interface Controller is no longer locked. A ring break is not necessarily the inevitable conclusion of this error.
- **Network Change Event:** One of the nodes in the network has activated or deactivated its bypass, which means it “disappears” or “appears” as a new node.
- **Low Voltage:** The voltage of one or more devices is too low to maintain operation of the NetInterface (see 3.1.5.5 Undervoltage Management).

For the handling of these errors, there are the following general rules:

- **Local Handling of Errors:** Every device is responsible to handle every recognized error locally. Only the NetworkMaster handles errors for the entire network.
- **No Error Reporting:** For keeping error management simple, robust, and not error-prone, the devices do not report these errors.
- **Securing Streaming Signals:** Streaming sinks supervise the validity of their output (see 3.2.8.2.6 Supervising Streaming Connections).

  The streaming connections on the Network are not removed, except in case of a fatal error or a Network Change Event, which leads to the NetworkMaster sending out Configuration.Status(NotOK).

3.1.5.1 Fatal Error

A “fatal error” is a kind of error that prevents the modulated signal from being handed on in the ring. There are four possible reasons:

- A device has no or insufficient voltage.
- A modulated signal receiver is defective.
- A modulated signal transmitter is defective.
- The modulated signal connection between transmitter and receiver is interrupted.

3.1.5.1.1 Handling of Modulated Signal Off

If a device detected the presence of the Shutdown Flag and recognizes at its input that the modulated signal was switched off, it switches off its own output within $t_{ShutDown}$. In case there is the need to wake the network again, it has to wait for $t_{Restart}$.

If the modulated signal is switched off without a ShutDown.Start(Execute), there may be two causes (see *3.1.4.2 Detection of Sudden Signal Off and Critical Unlock*):

- Fatal error
- A device runs error handling (e.g., Critical Unlock). In such a case, the PowerMaster switches on the modulated signal again after $t_{Restart}$ has expired, if the vehicle’s status requires it. It wakes the network in the normal way and by that a re-initialization is done.

In case the presence of the Shutdown Flag is not detected, the device has to perform the shutdown procedure as described in section 3.1.4.2 (Figure 3-21).

If the modulated signal is switched off, in some cases it might be switched on again after a short time. If the application performs a shutdown immediately, some devices might need a long time to return to normal operation. Therefore, the application has to be prepared for Shutdown, but has to stay active for $t_{PwrSwitchOffDelay}$. If the modulated signal reappears within $t_{PwrSwitchOffDelay}$, the system is re-initialized like when waking up after sleep mode. The only difference is that—within the devices—power supply, micro controller, and operating system need not be re-initialized.

3.1.5.1.2 Waking

If waking the network fails due to a fatal error and “modulated signal on” does not propagate through the entire ring, the Network Service changes to state NetInterface Off after t_{Config}.

The node waits for $t_{Restart}$ and then tries again to wake the network (1st retry is mandatory). Two additional retries are allowed (a total of three retries).

Independently, the PowerMaster is allowed to continue waking up the network as long as the condition to wake the network is fulfilled.

3.1.5.1.3 Operation

If there is a fatal error during normal operation, “modulated signal off” propagates through the entire ring. This is handled as described above. In case the power status of the vehicle requires it, the PowerMaster tries to wake the network after $t_{Restart}$. So the handling of a fatal error during waking needs to be performed (see above).

3.1.5.2 Unlock

An unlock occurs when a Slave device cannot lock onto the modulated signal of the MOST Network.

Causes for this may be that two TimingMasters in one ring are working against each other or if a TimingMaster does not receive a comprehensible signal.

Another cause can be that the modulated signal at a node's input is too weak or a node opens or closes its bypass. Every node downstream from the location that caused the unlock, up to the TimingMaster, recognizes the unlock. The nodes downstream of the TimingMaster up to the location that caused the unlock do not recognize the unlock. On an unlock, data errors occur. Based on its securing mechanism, the Control Channel is relatively insensitive to short unlocks.

If an unlock occurs, all synchronous data sinks (e.g., amplifiers) must secure their output signals; the Mute property remains unchanged

The Network Service checks the length of an unlock or the occurrence of a series of unlocks. If the length of a single unlock exceeds the time t_{Unlock}, an Error Shutdown event (Critical Unlock) is generated.

In case of a series of unlocks, the time of the different unlocks are accumulated. If this accumulated time is greater than t_{Unlock} (a single unlock which causes a Critical Unlock), an Error Shutdown event is generated. The accumulated time is reset whenever a Stable Lock is reached, that is, if there is a lock that lasts at least t_{Lock}.

The following example will clarify the meaning. (The timer values used can be found in section 3.2.9 on page 197).

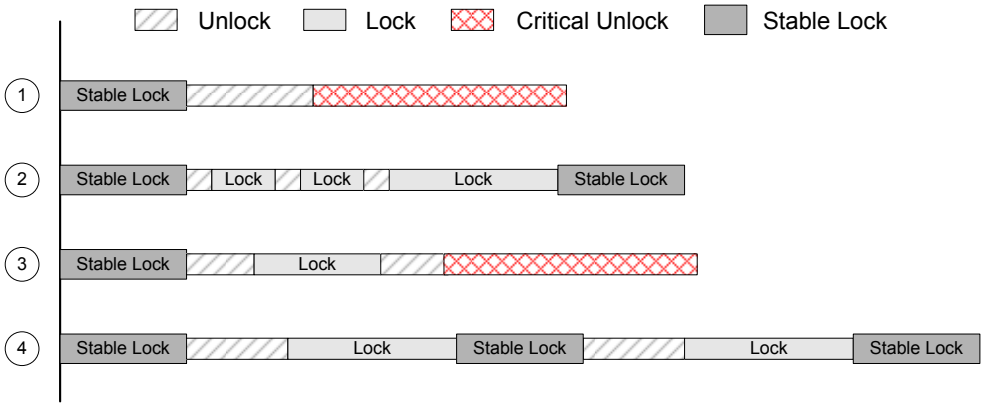

Figure 3-22: Examples of the behavior when unlocks occur.

1. The first example shows an unlock that persists longer than t_{Unlock}. This results in a Critical Unlock (Error Shutdown event).
2. A series of short unlocks with an accumulated time less than t_{Unlock} will not lead to a Critical Unlock.
3. Two unlocks with an accumulated time that exceeds t_{Unlock}. This leads to a Critical Unlock.
4. The unlocks are almost as long as t_{Unlock}. The example shows that the system can withstand a series of long unlocks, provided that a lock time of at least t_{Lock} is interspersed.

In addition, the change in number of MOST devices is checked. If this number indicates a Network Change Event, the application will be informed.

3.1.5.3 Network Change Event

A Network Change Event (NCE) is defined as a detected change of the Maximum Position Information transmitted cyclically on the Network.

If a device opens or closes its bypass, that is, enters or leaves the Network, the Maximum Position Information changes (except for the case when one device enters and another device leaves the network in a very short time interval[1]).

Disturbances of the Maximum Position Information can occur, for example, because of an unlock.

An NCE is recognized by the Network Service in every device.

If an additional node joins the network, the new node must be integrated on system level. Therefore, SystemCommunicationInit must run (refer to section 3.1.3.1.1.2). In order to achieve that, the NetworkMaster checks configuration again and broadcasts Configuration.Status.

If a node leaves the network, any sink that was connected to that node must secure its output signal immediately. The Mute property is set to “On”. Furthermore, every node must be able to handle the case where a communication partner is missing and must act accordingly in a safe way. The NetworkMaster checks configuration again and broadcasts Configuration.Status.

3.1.5.4 Failure of a NetworkSlave Device

The failure of a NetworkSlave device can be divided into two scenarios:

1. **A Network Interface Controller failure**, which leads to a reset of the whole device; that is, Network Interface Controller, application etc. (bottom-up reset).

2. **An Application failure**. The Network Interface Controller is still OK, hence only the affected applications need to be restarted.

3.1.5.4.1 Failure of the Network Interface Controller

If a device experiences an internal failure of the Network Interface Controller the whole device must be re-initialized.
Note that the device must not communicate until it receives a Configuration.Status message.

3.1.5.4.2 Failure of an Application

Every application must be able to handle the case where one of its communication partners does not respond and safely terminate the parts of the program that depend on this communication.

By implementing a watchdog in each device, a long “hanging” of an application should be avoided.
It may happen that single processes in a device are hanging (but not the entire device, as in section 3.1.5.4.1, 'Failure of the Network Interface Controller’), and that those processes need to be restarted. In case this failure stops an entire, or even several FBlocks, the device has to un-register those FBlocks in the Central Registry. This is done through a notification of the new status of FBlockIDs sent to the NetworkMaster (NWM):

```
Device -> NWM: NetBlock.RxTxLog.FBlockIDs.Status (FBlockIDList)
```

[1] typically up to 24 ms

This states that only those FBlocks contained in FBlockIDList are available. The NetworkMaster updates the Central Registry and broadcasts immediately after the reception of such an un-registration:

```
NWM -> All: NetworkMaster.1.Configuration.Status  (Control=Invalid,
                                                    DeltaFBlockIDList)

Control              Unsigned Byte       0: NotOK
                                         1: OK
                                         2: Invalid
                                         3: Reserved
                                         4: NewExt

DeltaFBlockIDList                 List of FBlockID.InstID
```

A detailed description of the handling of “Control = Invalid” and “Control = NewExt” is to be found in sections 3.1.3.3.6 and 3.1.3.4.3.

DeltaFBlockIDList is the list of those FBlocks that are invalid. Therefore, all applications have the required information and can terminate functions depending on the invalid FBlocks.

When the failed process is ready (after being killed and re-initialized), the depending FBlocks are registered again:

```
Device -> NWM: NetBlock.RxTxLog.FBlockIDs.Status (FBlockIDList)
```

However, the device must not attempt to register its FBlocks again before having successfully transmitted its deregistration message to the node containing the NetworkMaster.

Immediately upon reception of the registration message, the NetworkMaster registers these FBlocks in the Central Registry and broadcasts:

```
NWM -> All: NetworkMaster.1.Configuration.Status (Control=NewExt,
                                                   DeltaFBlockIDList)
```

Here, DeltaFBlockIDList is the list of the new FBlocks.

The NetworkMaster has to process FBlockIDs.Status messages in the order they are received.

Note: *In case a device that starts up fast has single FBlocks starting up relatively slow, the same mechanism of supplementary registration can be used. However, the status message may not be sent before the NetworkMaster has asked the device.*

For details of the Central Registry update behavior, please refer to sections 3.1.3.3.6.1 and 3.1.3.3.6.2.

3.1.5.5 Undervoltage Management

Undervoltage conditions do not inevitably occur in every device at the same time and in the same severity. There are two limits regarding the supply voltage of a device:

Critical voltage $U_{Critical}$:
First, there is the limit at which the application might no longer work safely but where communication is still possible. In that case, a source must route zeros and a sink must secure its output signals. The Mute property remains unchanged. In case of a recovery, the output signals can be restored immediately.

The System Integrator may determine additional measures.

Low voltage U_{Low}:
Second, there is a device specific limit, where even the NetInterface no longer works reliably, so even communication cannot be maintained.
If U_{Low} is reached, the device switches off the modulated signal and switches to DevicePowerOff Mode. The device stays in DevicePowerOff mode, even if the supply voltage recovers. It is awakened either by "modulated signal on" at its input, or by the demand for communication from its own application. It changes to mode DeviceNormalOperation via the standard initialization process.

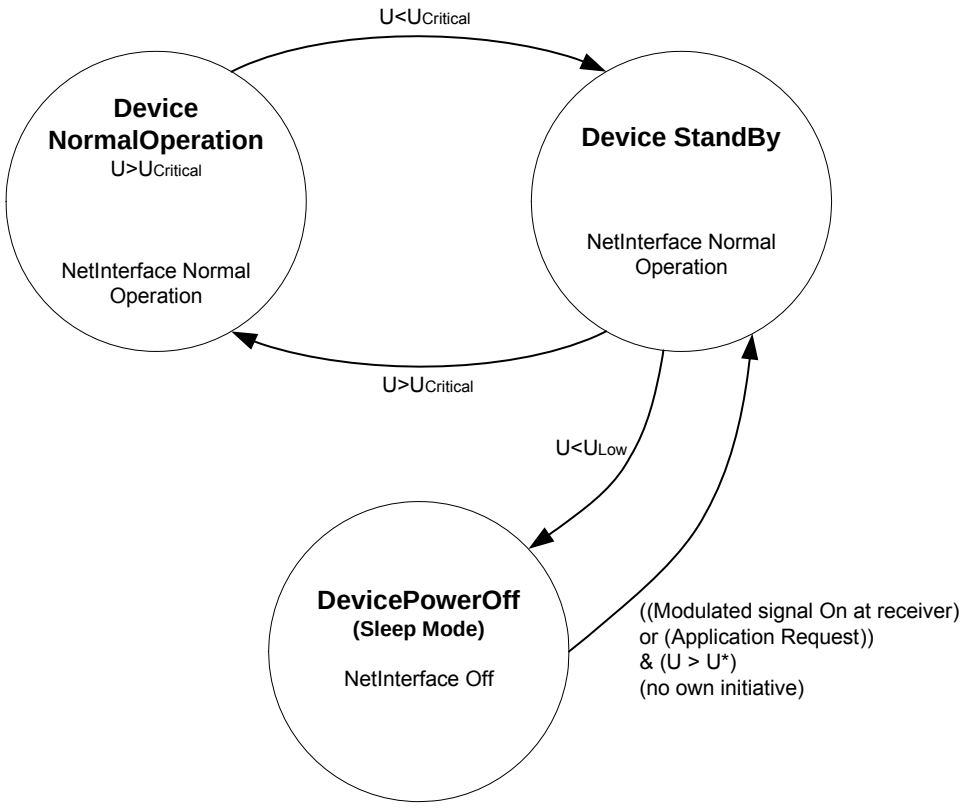

Figure 3-23: Behavior of a device depending on supply voltage[1]

[1] ***Notes:***
1. *It is up to the System Integrator to decide whether an application is powered or not.*
2. *U_{Low} and $U_{Critical}$ are defined by the System Integrator and have corresponding hysteresis values.*
3. *U* is either U_{Low} or $U_{Critical}$ under consideration of the corresponding hysteresis value.*

3.1.5.6 Over-Temperature Management

Some devices could experience malfunctions or permanent damage when exposed to temperature conditions above their operating limits. Even though it should be the design goal of every system that such condition is never reached during normal operation, it is still necessary to define the system's behavior for this worst case.

This section defines the behavior of the PowerMaster, which must be able to react to over-temperature conditions of its Slaves. It also applies to every Slave device that can monitor its own temperature and decide when to take appropriate action. Such a Slave has to implement the mandatory procedures as described below and may, in addition, support any combination of optional over-temperature measures.

After a system shutdown that was caused by over-temperature, the temperature should sink to a level that guarantees a reasonable amount of operation time when the system is back up again, that is, it is not very useful to have a system that restarts and remains in operational state for just a minute.

Different strategies are presented for the restart of the system after an over-temperature shutdown; they work independent of each other and can even be mixed within one system, if desired.

3.1.5.6.1 Levels of Temperature Alert

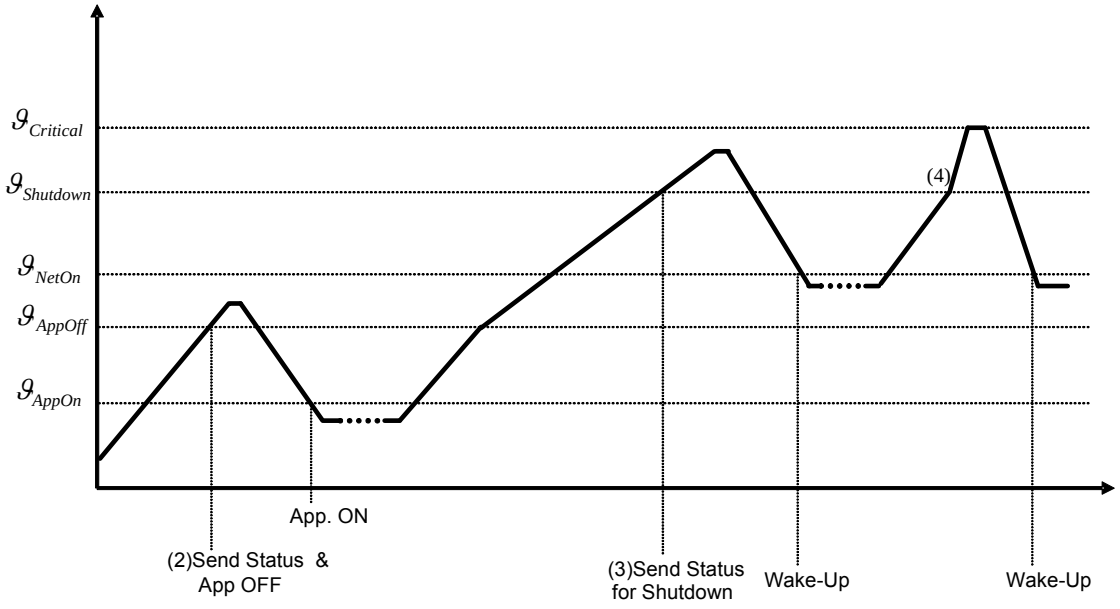

Figure 3-24: Alert levels

Four different temperature alert levels can be identified. From least severe to most severe, the levels are:

1. Limited application functionality level[1]
2. Individual application shutdown level (ϑ_{AppOff})
3. Temperature shutdown level ($\vartheta_{Shutdown}$)
4. Critical temperature level ($\vartheta_{Critical}$)

Figure 3-24 shows three of the four different temperature alert levels.

[1] Not shown in the figure.

Note that all temperature levels (those for the alerts, as well as those for restart) are device-specific and are handled on a device-internal basis.

No central component in the MOST network supervises the temperature of a device and decides for the device when it has to shut down; this is completely at the device's discretion.

3.1.5.6.2 Mandatory Over-temperature Behavior

After the system is shut down, it has to stay switched off long enough for the device to cool down.
This means that after an over-temperature shutdown, the PowerMaster must not restart the system until $t_{WaitAfterOvertempShutDown}$ time has passed[1].

3.1.5.6.2.1 Mandatory Slave Behavior

Temperature shutdown level

If the temperature exceeds $\vartheta_{Shutdown}$, the device must request a temperature shutdown from the PowerMaster. This is done by broadcasting NetBlock.ShutDown.Result with parameter 0x03 (Temperature Shutdown).

Critical temperature level

If the critical temperature ($\vartheta_{Critical}$) is reached, an immediate shutdown is initiated by the device, which is in critical condition, by simply switching the modulated signal off.

Restart

If during a restart attempt the device finds that it is still above the restart temperature threshold, it broadcasts NetBlock.ShutDown.Result(0x03) again immediately after the NetOn state is reached. After successful transmission, the device switches off the modulated signal.

3.1.5.6.2.2 Mandatory PowerMaster Behavior

When the PowerMaster receives NetBlock.ShutDown.Result(0x03) from a Slave device, the PowerMaster recognizes that the Slave device is experiencing an over-temperature situation.

If modulated signal is switched off by the device with the over-temperature condition because it has reached $\vartheta_{Critical}$, the PowerMaster shall avoid an immediate restart of the network. The PowerMaster is considered to be in "over-temperature-mode", a state that is maintained beyond the shutdown of the system.

Restart

A minimum time between restart attempts of the system shall be guaranteed so the device has a chance to cool down. A failing attempt to restart the network lasts no longer than approx. 150ms, so if the minimum interval between such attempts ($t_{WaitAfterOvertempShutDown}$) is, for example, one minute, the short phase of operation under the persisting over-temperature condition can be considered insignificant.

The PowerMaster has to support one of the following restart triggers:

a) The PowerMaster decides, after a while, to attempt a restart. It wakes up the MOST system.

b) The PowerMaster is triggered to restart the ring upon user request.

[1] As described later, the system may be restarted by the overheated device before $t_{WaitAfterOvertempShutDown}$ expires if the device has the capability to monitor its own temperature during the cooling phase.

If the PowerMaster receives NetBlock.Shutdown.Result(0x03) after system wake-up, it shuts down the system again (without the standard procedure).

If at restart the NetworkMaster reaches System State OK, the over-temperature condition of the system is over.

3.1.5.6.3 Optional Over-Temperature Behavior

3.1.5.6.3.1 Optional Slave Behavior

Limited application functionality level
Above a certain temperature, an application may decide to limit its functionality in order to reduce power dissipation and, hence, the warming up of the device. This could be done “silently” by the application or with an appropriate notification of the application’s Controller. An example is volume limitation in order to reduce the power stage’s power dissipation.

Individual application shutdown level
If, for example, a telephone unit becomes warmer than its maximum operating temperature (which is still below the maximum operating temperature for the Physical Interface unit or other components needed for MOST functionality), the device could decide to shut down this specific application. The FBlock is removed from the Central Registry after sending an updated NetBlock.FBlockIDs.Status message to the NetworkMaster.

Restart
If the device is able to supervise its own cooling phase, it may wake up the ring when it has cooled down.

3.1.5.6.3.2 Optional PowerMaster Behavior

After receiving NetBlock.Shutdown.Result with parameter 0x03 (Temperature Shutdown) from the Slave that experiences the over-temperature condition, the PowerMaster may execute the standard shutdown procedure as described under section 3.1.2.3.2.

Note: *This entails that the normal shutdown procedure may not finish due to another device constantly sending NetBlock.Shutdown.Result(Suspend), eventually forcing the device with the over-temperature condition into the critical temperature level.*

3.2 Network Layer Protocol Specification

The Network Layer protocol is the set of rules that governs the format and meaning of the information exchange between the Network Layers in various devices. The Network Layer uses the protocol to implement the Network Layer services definitions.
The protocol machine defines the various states of an Network Layer and the valid transitions between the states. It may be considered as a finite state machine. The protocol machine exchanges information with other protocol machines.

3.2.1 Support at System Startup

The MOST Network Interface Controller provides mechanisms for system startup. All components of the system get a unique number (physical position value), with numbering starting at the TimingMaster at 0x00, and then incremented by one. These numbers are used for node position addressing. Furthermore, every device receives the information about the total number of devices in the ring. The MOST Network Interface Controller also provides a wake-up mechanism.

3.2.2 Addressing

MOST addresses are divided into two categories: 16 bit and 48 bit addresses.

FUNCTIONS				
FktID	**OPType**	**Sender**	**Receiver**	**Explanation**
NodePosition Address	Get	Controller	NetBlock	Requesting node position address
	Status	NetBlock	Controller	Answer
NodeAddress	Get	Controller	NetBlock	Requesting logical node address
	Status	NetBlock	Controller	Answer
GroupAddress	Get	Controller	NetBlock	Requesting group address
	Status	NetBlock	Controller	Answer
	Set	Controller	NetBlock	Setting group address
FBlockIDs	Get	NetworkMaster	NetBlock	Requesting list of functional addresses (FBlockID.InstID combinations)
	SetGet	NetworkMaster	NetBlock	Seting a new InstID for an FBlock
	Status	NetBlock	Controller	List of FBlockID.InstID combinations implemented by the node
EUI48	Get	Controller	NetBlock	Requesting MAC address
	Status	NetBlock	Controller	Answer

Table 3-16: Functions in NetBlock that handle addresses

3.2.2.1 16 Bit Addressing

Node Position Address
A node position address is unique by definition but node position addressing is not used under normal operation conditions. It is used only for administrative tasks, for example, by the NetworkMaster during initialization.

A node position address can be determined using the **NodePositionAddress** function in the NetBlock. It consists of an offset plus the physical position value:

RxTxPos = 0x0400 + Pos

Pos = 0x00 for TimingMaster
Pos = 0x01 for first device in ring...

Logical Node Address

Logical node addressing is used by all nodes to address a single node. Logical node addresses can be dynamic or static.
The logical node address can be requested from the function **NodeAddress** in the NetBlock.

Dynamic logical node address

A logical node address must be unique even if there are multiple devices of the same type. Therefore, it is derived from the unique node position address. During initialization of the network, the logical node address is calculated by each device as follows:

RxTxLog = 0x0100 + Pos

The device containing the TimingMaster is located at physical position 0; it is assigned with the logical node address 0x0100. A device at position five in the ring will have address 0x0105.

The dynamic logical node address is recalculated on each transition to NetInterface Normal Operation.

Static logical node address

One approach for assigning a logical node address in the static address range is to assign certain address ranges with respect to the functionality of devices. This means, for example, that the first video display module in a network gets address 0x0200, the second 0x0201, etc., while the first active amplifier gets address 0x0188.

Group Address

The group address can be requested from function GroupAddress in the NetBlock and can be modified using this function if required. The default procedure for deriving a group address is to take the FBlockID of the FBlock that is most characteristic for the device. The high byte of the group address is always fixed to 0x03:

GroupAddress = 0x0300 + FBlockID

The FBlockID that is reported first in case of a request for the FBlockIDs is typically the most descriptive for the device.

Groups can be built dynamically by modifying group addresses.

Group addressing is typically used for controlling several devices of the same type (e.g., active speakers). The grouping of devices must be established during the definition of the system.

Broadcast Address

Broadcast addressing requires a great deal of system resources and, therefore, should be used for administrative tasks only.

- A single transfer or segmented message that is sent to the blocking broadcast address (0x03C8) is received by all nodes in the ring. Until the last node in the ring has acknowledged a broadcast message, communication via the Control Channel is suppressed for other messages. This mechanism guarantees that all devices receive important broadcast messages.
  The removal of the block is initiated by the sender of the blocked broadcast message. On data link layer, this is done by transmitting a free-up message. The free-up message is sent to address 0x0000. It is processed in the Network Interface Controller and is never handed over to the Application Layer.
- An unblocking broadcast message (either single transfer or segmented transfer to address 0x03FF) does not block the transmission of other control messages. This kind of transmission can be used for uncritical data transmission, which might not necessarily be received by all devices.

16 bit address mapping

The different ways of 16 bit addressing are mapped into the address area of a MOST Network Interface Controller:

Address range	Mode
0x0000	Internal Communication. This address can also be used for administrative communication between Network Interface Controllers on Data Link Layer. Messages transmitted over the MOST network using 0x0000 as target address are not handed over to the Application Layer.
0x0001…0x000F	Internal Communication
0x0010…0x00FF	Static address range
0x0100…0x013F	Dynamic calculated (0x0100+POS) address range
0x0140…0x02FF	Static address range
0x0300…0x03FF	Reserved for group/ broadcast
0x0400…0x043F	Node position (0x0400 + POS) address range.
0x0440...0x04FF	Reserved
0x0500…0x0FEF	Static address range
0x0FF0	Optional debug address
0x0FF1…0x0FFD	Reserved
0x0FFE	Init address of Network Service
0x0FFF	Init address of Network Interface Controller
0x1000…0xFFFE	Reserved for future use
0xFFFF	Un-initialized logical node address

Table 3-17: Addressing modes vs. address range

3.2.2.2 48 Bit Addressing

Ethernet MAC Address
The permanent unique address is an EUI-48, conforming to the IEEE Standard. The EUI-48 can be used as Ethernet compatible MAC address. Each MAC address is unique and has a length of 48 bits.

It is in the responsibility of each manufacturer to acquire its own OUI resp. block to build EUI-48 and MAC addresses[1].

48 bit addressing supports the Ethernet-typical address comparison modes:

- Unicast (Perfect Filtering, 48 bit match)
- Multicast (Hash Filtering)
- Broadcast (0xFFFF FFFF FFFF)
- Promiscuous (unfiltered)

It is not required that each device supports all 48 bit comparison modes.
The selected mode is set at the MOST Network Interface Controller.

3.2.3 Low-Level Retries

In case the sending of a Control Message is not successful, the MOST Network Interface Controller can re-send the message automatically. The MOST Network Interface Controller may allow specifying the number of retries and the delay between the retries. Typically, these values should not be changed; however, they can be modified to fine tune the system.

While low-level-retries are performed, it is guaranteed that groupcast and broadcast messages are only received once in every node.
However, it is possible that the same groupcast or broadcast message may be received more than once caused by retransmissions on higher levels.

[1] See IEEE 802.3—Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications.

3.2.4 Handling Overload in a Message Receiver

The MOST Network Interface Controller informs the sender's Network Service by a NAK error message indicating that the receiving node has rejected a telegram although the low-level retries were used. This is an indicator for a momentary overload or a defect. The Network Service passes the NAK error message through to the application, which has to decide what needs to be done (retry, reject, postpone).

If that telegram belongs to a connection where data is sent continuously from a sender to a receiver, an optional mechanism can be implemented which adapts the telegram transfer rate to the speed of the message receiver. A simple mechanism may look like this:

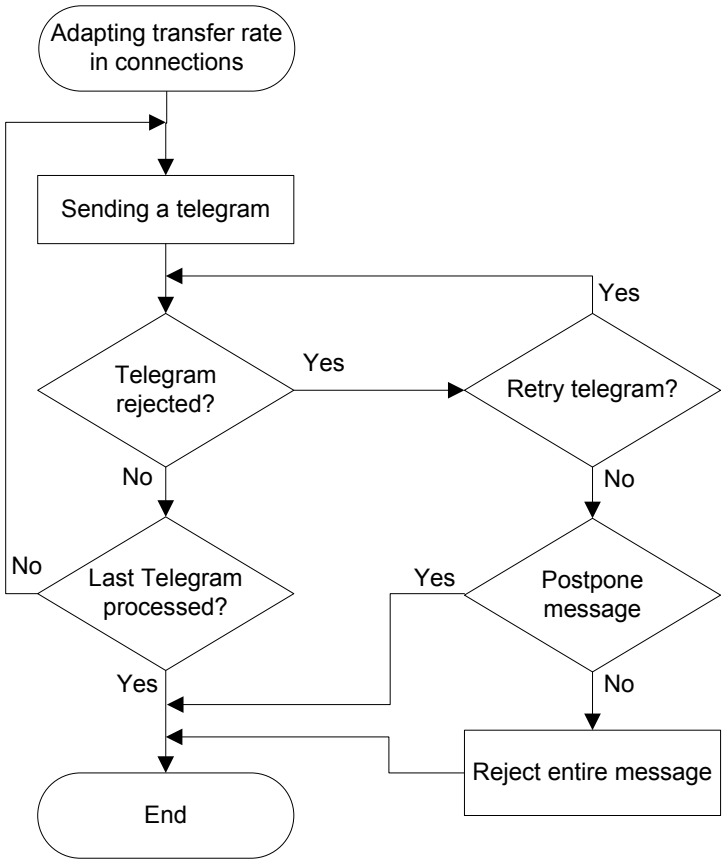

Figure 3-25: Possible mechanism to adapt transfer rates to the speed of a message receiver

It is assumed here that errors due to incorrect address or CRC errors are handled "on top" of that mechanism. "Message" refers to the entire amount of data to be sent. A telegram is that portion of data which can be transported on the Control Channel. It transports a part of the entire message. Rejecting a telegram means that the target node could not process it due to an occupied receive buffer. In that case, the MOST Network Interface Controller has already run its low-level retries. Now the application has three selections:

1. The telegram can be sent again, thus having additional low-level retries available.
2. The entire message can be rejected, for example, because it is no longer relevant.
3. The entire message can be postponed, that is, sent later.

3.2.5 MOST Message Services

3.2.5.1 Control Message Service

Via the MOST Network Interface Controller, MOST telegrams can be sent and received. MOST telegrams consist of a sender or receiver address respectively (RxTxAdr), and a maximum number of L_{CMSmax} data bytes [1].

		Data area of MOST Network Interface Controller				
16 bits	16 bits	8 bits	8 bits	8 bits		8 bits
Destination Address	Source Address	Data 0	Data 1	Data 2	…	Data L_{CMSMax} -1

The Network Service provides a mechanism which is called Control Message Service (CMS). It handles the setting and reading of the registers of the MOST Network Interface Controller.

[1] L_{CMSmax} is speed grade dependent. In MOST150 systems, for example, L_{CMSmax} is 51.

3.2.5.2 Application Message Service (AMS)

3.2.5.2.1 AMS Protocol Description

MOST Network Service provides two different types of transmissions via the Control Channel. These are mandatory for each device:

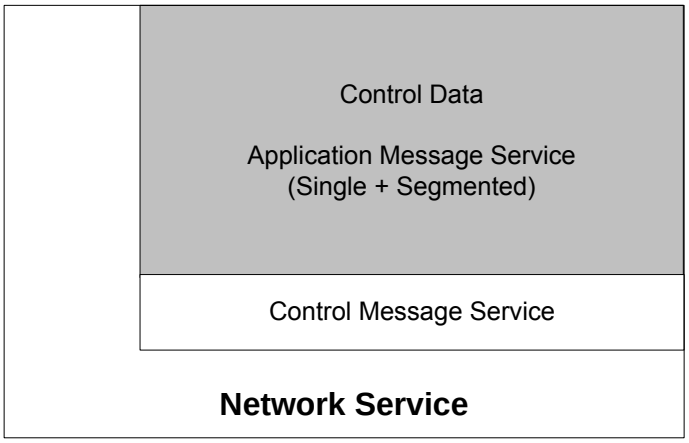

Figure 3-26: Network Service: Services for Control Channel

- **Single transfer:** Data packets that do not exceed $L_{AMSmax} = L_{CMSmax} – 6$ bytes (of the L_{CMSmax} available CMS data bytes, a total of 6 bytes is reserved for Message ID, TelID and TelLen) are transmitted in a single transfer.
- **Segmented transfer:** Commands and status messages with a length greater than L_{AMSmax} bytes are transported by multiple telegrams. These segmented transfers have a maximum length of 65,535 bytes.

As already described in section 2.2.1 on page 40, messages of the following type must be transmitted:

```
DeviceID.FBlockID.InstID.FktID.OPType (Parameter)
```

The Application Message Service (AMS) is based on the Control Message Service (CMS). MOST telegrams transport application messages. Each telegram is divided as outlined in the following diagrams.
Please note that the destination address is not forwarded to higher layers. The information that is passed on, only indicates whether a message was received by single cast or multi cast.

The Message ID is not entirely transparent to the Application Message Service: AMS will set the last 4 bits of the Message ID to 0xF in a case of a Segmentation Error.

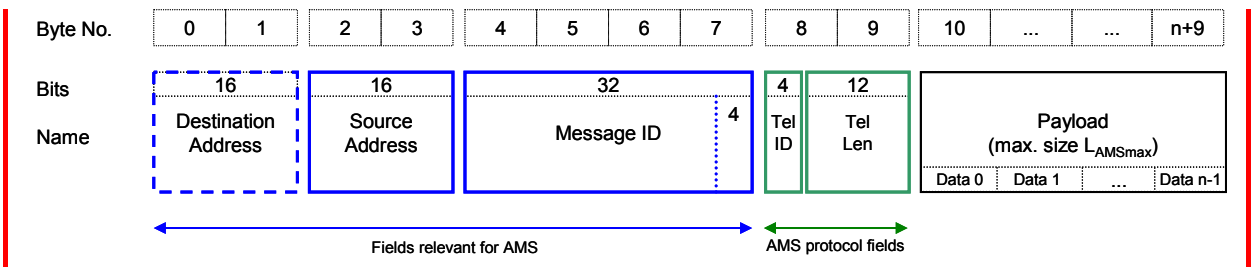

Figure 3-27: Single transfer (n bytes payload)

Segmented Transfer

The first segment of a segmented transfer has TelID 1; then 0 or more TelID 2 segments follow. The final segment has TelID 3. Compared to single transfer, the maximum payload is reduced by one byte that is used by the message counter (MsgCnt).

For segments with TelID 3, the available payload must not be entirely unused, that is, TelLen must be greater than 0.

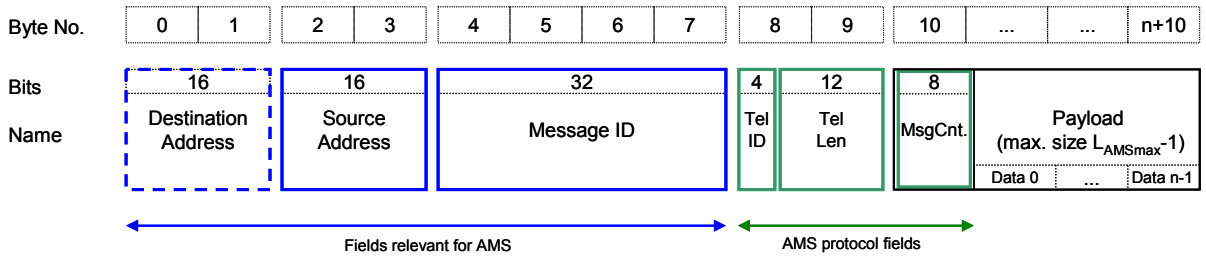

Figure 3-28: One telegram (n bytes payload) of a segmented transfer

Note: *The receiver has to accept telegrams with TelID 1 or 2 that do not use the entire available payload.*

Figure 3-29 is an example of a segmented transfer in a MOST150 system where the entire available payload is used in the first telegram (TelID=1, MsgCnt=0x00) and the second telegram (TelID=2, MsgCnt=0x01). For MOST150, the maximum available payload (L_{AMSmax}) is 45; with one byte of the payload being used by the message counter, 44 data bytes are available.

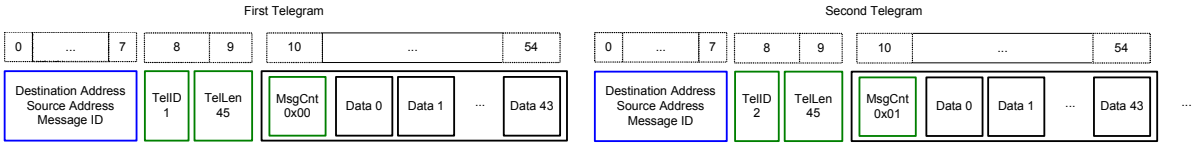

Figure 3-29: Segmented transfer example with entire available payload used

Figure 3-30 depicts a MOST150 segmented transfer where the available payload for the first and second telegram is not used entirely. Of the available 44 data bytes for the first telegram only 2 are used. The second telegram only uses 18 data bytes where 44 would be available.

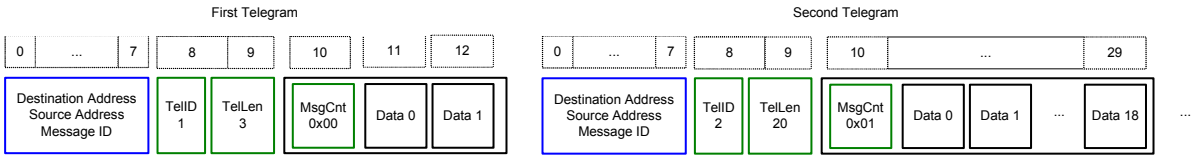

Figure 3-30: Segmented transfer example with available payload not entirely used

Unless size-prefixed segmented transfer is used (see below), the length of the application message is not transmitted directly. It has no meaning on telegram level, since several telegrams may be required to transport one message. Nevertheless, length is transmitted indirectly via TelLen and MsgCnt and must be restored on the receiver's side.
MsgCnt will start at 0x00 for the first telegram and increase by 1 with each following telegram. When 0xFF is reached, counting restarts at 0x00. This is repeated until the entire message is sent.

Individual telegrams of a segmented message must not be more than $t_{WaitForNextSegment}$ apart; otherwise, an error is reported (ErrorCode 0x0C, ErrorInfo 0x05).

Messages from one node to the same address (RxTxAdr.FBlockID.InstID.FktID.OPType) must not be interleaved. This means that during an ongoing segmented transfer any attempt by the sending node to transmit a message with an identical signature will result in an error (ErrorCode 0x0C, ErrorInfo 0x07).

For the complete list of segmentation errors, see *Table 2-5: ErrorCodes and additional information (part 1)*.

Size-Prefixed Segmented Transfer

In its first segment (identified by TelID 4), the size-prefixed segmented message contains no payload apart from the message size (total payload) in bytes. This allows the receiver to allocate the exact amount of buffer space before the first payload chunk arrives.
The remaining available bytes of the first telegram are reserved for future use.
The remainder of the message is transmitted in the manner of a regular segmented transfer: A TelID 1 segment is followed by 0 or more TelID 2 segments. A TelID 3 segment completes the message.

If a device receives an application message with TelID 4 that has an invalid length (i.e., TelLen differs from 2), the devices behaves as if it had not received the message at all; that is, no allocation of buffer space is triggered and no error message is sent to the originator.

MOST nodes must not send size-prefixed segmented messages with message size 0 ≤ MsgSize ≤ L_{AMSmax}. If, for example, in case of an error of the sender, a device receives an application message with TelID 4 that has an implausible value in the message size field (0 ≤ MsgSize ≤ L_{AMSmax}), the message is ignored; no error message is sent to the originator of the message with the unexpected MsgSize.

It is recommended that within size-prefixed segmented messages the segments with TelID 1 or 2 use the entire available payload, that is, TelLen is L_{AMSmax}.

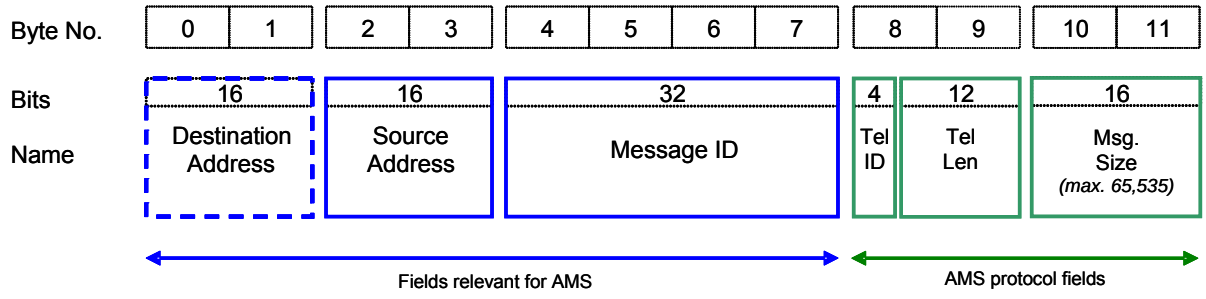

Figure 3-31: First telegram of a size-prefixed segmented message

TelID, TelLen, and Data

The telegram length (TelLen) is not an upper bound for the data length instead it is mandatory that the TelLen is the exact length of the valid data.

TelID: Identification of kind of telegram

Meaning	TelID	Data 0 = MsgCnt
Single Transfer	0	Data 0
1st telegram segmented transfer	1	0x00
2nd telegram segmented transfer	2	0x01
…	2	…
…	2	0xFF
…	2	0x00
…	2	…
(n-1)th telegram segmented transfer	2	0x(n-1)
Last telegram segmented transfer	3	0xn
Size-prefixed segmented message	4	Msg. Size

Table 3-18: Use of the TelID field

If a device receives an application message that contains a TelID outside the valid range (i.e., TelID greater than 4), the device ignores the message; no error message is sent to the originator of the invalid message.

TelLen: 0...L_{AMSmax}; the number of data bytes that are valid in the telegram.

Data 0-Data L_{AMSmax} -1**:** Data bytes

3.2.6 Handling Packet Data

The data field of the Packet Data Channel is significantly longer than that of the Control Channel. Additionally, a transport protection must be implemented on a higher level.

Note: *The Packet Data Channel can also be used for the transmission of control data.*

3.2.6.1 MOST Network Service

For the transport of packet data, The Network Service has a mechanism, called the Packet Data Transmission Service.

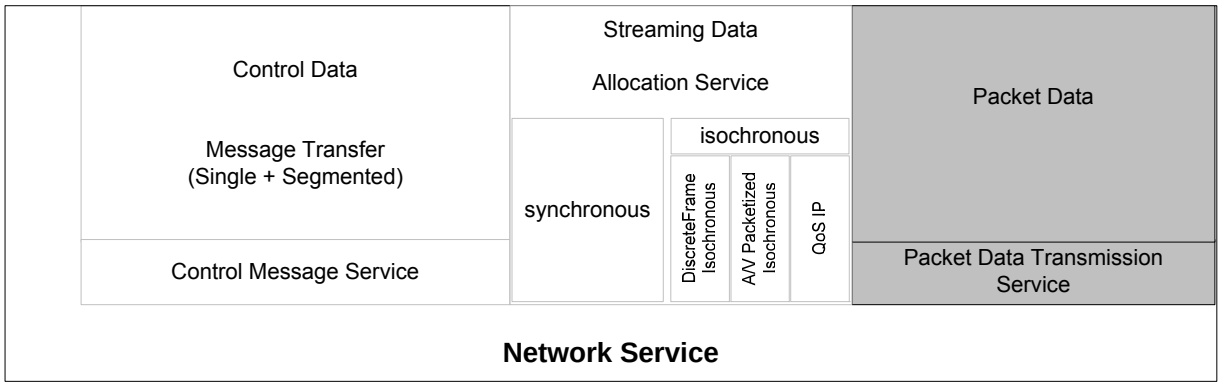

Figure 3-32: Network Service—services for the Packet Data Channel

3.2.6.1.1 Use Cases and Data Formats

The Packet Data Channel provides the following use cases

- MOST High (16 bit addressing)
- Ethernet over MOST (48 bit addressing)

3.2.6.1.1.1 MOST High

For connections, that is, the transmission of data streams or the transmission of larger data packets, a higher transport protocol, the MOST High Protocol can be used, which is derived from the well-known Transport Control Protocol (TCP). It uses some of the mechanisms defined by TCP but can only be used for communication within the MOST network. MOST High Protocol transports data and could be used for transporting data coming from the external world (GSM) that is secured by the “real” TCP.

Data Area MOST Network Interface Controller = max. 1524 bytes (16 bit addressing)

16 bits	16 bits	8 bits	8 bits	12 bits	4 bits	4 bits	12 bits	8 bits	8 bits		8 bits
Destination Address	Source Address	FBlock ID	Inst. ID	Fkt ID	OP Type	Tel ID	Tel Len	Data 6	Data 7	...	Data 1523

TelID: Identification of telegram type

Meaning	TelID
MOST High Protocol User Data	8
MOST High Protocol Control Data	9

TelLen: up to 1518
Specifies the length of the data field, that is, the number of bytes after TelLen

Data X: Data bytes

3.2.6.1.1.2 Ethernet over MOST

Data Area MOST Network Interface Controller = max. 1522 bytes (48 bit addressing)

48 bits	48 bits	8 bits	8 bits		8 bits	32 bits
Destination Address	Source Address	Data 0	Data 1	...	Data 1505	Packet Data Channel CRC

For the frame structure, refer to IEEE 802.3.

3.2.7 Handling Streaming Data

3.2.7.1 MOST Network Service API

The MOST Network Interface Controller provides mechanisms for administrating Streaming connections In addition, the Network Service provides an API to simplify the use of those mechanisms.

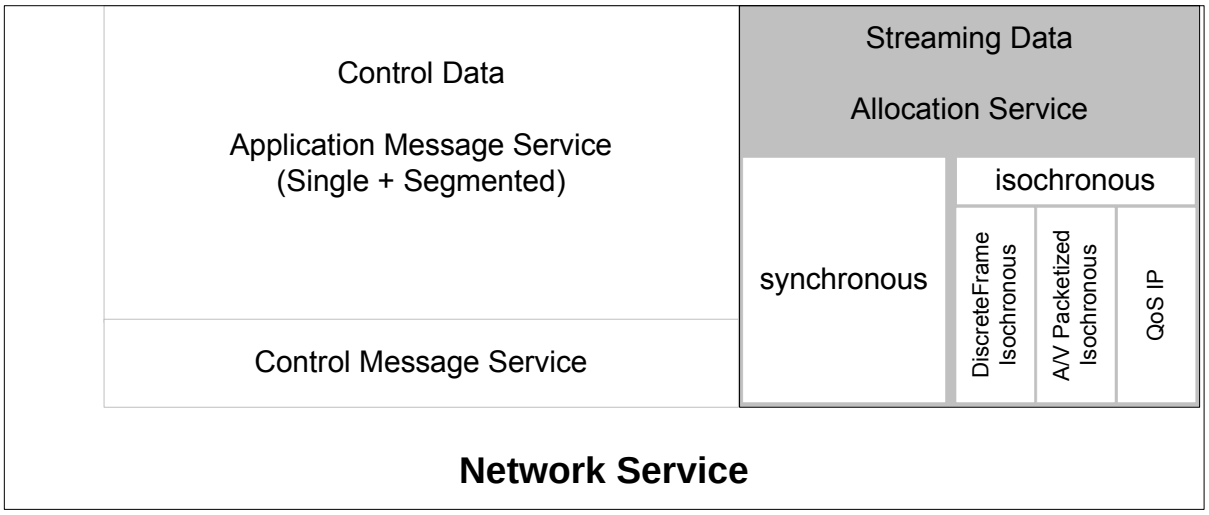

Figure 3-33: Network Service for streaming data

On the network, streaming data and packet data can be transported within a MOST frame.
Definition:

- **A streaming connection is characterized by a connection label and the required bandwidth.**

Bandwidth is allocated implicitly by creating a socket where the size of the socket denotes the amount of bandwidth. By simply destroying the socket, the allocated bandwidth is de-allocated.
This approach relies on a connection label (16-bit word) combined with bandwidth information (also 16-bit word), instead of a long list of single byte channels.
Furthermore, the routing and the low-level allocation are entirely encapsulated by the Network Interface Controller and of no influence to the application.

3.2.7.2 FBlock Functions

On the application level, different basic functions in sources and sinks are realized, which serve the administration of streaming connections. They themselves access the routines of the Network Service. The FBlock functions can be categorized into three categories:

- Functions that represent the whole device and are located in NetBlock.
- Functions that provide information about both sources and sinks within an FBlock.
- Functions that deal specifically with only sources or sinks within an FBlock.

3.2.7.2.1 General Source / Sink Information

An FBlock containing a source or sink provides the following function:

StreamDataInfo
Function StreamDataInfo can be used to get information on how many connections the FBlock may serve as source or as sink. A request is sent:

```
Controller -> Slave: FBlockID.InstID.StreamDataInfo.Get
```

The answer contains a list of the source and sink numbers in this FBlock:

```
Slave -> Controller: FBlockID.InstID.StreamDataInfo.Status (SourceNrList,
                                                            SinkNrList)
```

3.2.7.2.1.1 Streaming Source

Some of the functions that a streaming source provides to the network are shown below.

- **SourceInfo**
  Property SourceInfo contains detailed information about the kind of streaming source data that the source can handle. The source information is specific for each source number.

  On a request with the SourceNr:

  ```
  Controller -> Slave: FBlockID.InstID.SourceInfo.Get (SourceNr)
  ```

  The following is received:

  ```
  Slave -> Controller: FBlockID.InstID.SourceInfo.Status (SourceNr, BlockWidth,
                                  ConnectionLabel, [SourceDescription])
  ```

  SourceDescription determines the kind of data that is sent by the source.
  Information about streaming data types can be found in the corresponding Stream Transmission Specification.

- **SourceName**
  Property SourceName holds the name of the streaming source.
  It is requested with the SourceNr as parameter:

  ```
  Controller -> Slave: FBlockID.InstID.SourceName.Get (SourceNr)
  ```

  The answer is a string containing the name:

  ```
  Slave -> Controller: FBlockID.InstID.SourceName.Status (SourceNr, SourceName)
  ```

- **SourceActivity**[1]
  Some streaming source applications require either to start or to stop transmission of streaming data controlled by superior layers. In general, there are specific functions that need to be called for performing the starting or stopping. It is easier for the Controller if this specific information is not needed. Therefore, for every streaming source data stream, the method SourceActivity is defined.

  ```
  Controller -> Slave: FBlockID.InstID.SourceActivity.StartResultAck
                                          (SenderHandle, SourceNr, Activity)
  ```

  Through parameter Activity = [On/Off/Pause], streaming data transfer can be started, stopped, or paused.

  After completion of the respective action, the following reply will be generated:

  ```
  Slave -> Controller: FBlockID.InstID.SourceActivity.ResultAck
                                          (SenderHandle, SourceNr, Activity)
  ```

Routing and low-level allocation are entirely encapsulated by the Network Interface Controller and of no influence to the application. The approach for building streaming connections is called “Allocation approach”.

[1] SourceActivity is optional.

- **Allocate**
  To make the source first allocate bandwidth and then build a connection, method Allocate is used.

```
Controller -> Slave: FBlockID.InstID.Allocate.StartResultAck
                                                    (SenderHandle, SourceNr)
```

  On success, the bandwidth and connection label the source now occupies are reported.

```
Slave -> Controller: FBlockID.InstID.Allocate.ResultAck
                                                    (SenderHandle, SourceNr,
                                                BlockWidth, ConnectionLabel)
```

  If the allocation was not successful due to a lack of available bandwidth, an error is generated with the ErrorCode “Function specific” and as ErrorInfo the SourceNr and the required bandwidth. An allocation must never be done partially. Unless the bandwidth can be allocated completely, no allocation is done.
  The resulting error will be:

```
Slave -> Controller: FBlockID.InstID.Allocate.ErrorAck
                                          (SenderHandle, "Function Specific",
                                                  SourceNr, BlockWidth)
```

- **DeAllocate**
  DeAllocate is used by a Controller wishing to cancel that which was done by Allocate before. This means that the allocated bandwidth will be deallocated and that the source no longer is connected.

```
Controller -> Slave: FBlockID.InstID.DeAllocate.StartResultAck
                                                    (SenderHandle, SourceNr)
```

  On success, the connection is no longer occupied and the source is disconnected.

```
Slave -> Controller: FBlockID.InstID.DeAllocate.ResultAck
                                                    (SenderHandle, SourceNr)
```

3.2.7.2.1.2 Streaming Sink

An FBlock that is used as a sink for streaming data supports functions similar to those for a source. Error handling is also done in an analogous way.

Some of the functions that a streaming sink provides to the network:

- **SinkInfo**
  Property SinkInfo contains detailed information about the kind of streaming sink data that the sink can handle. The sink information is specific for each sink number.
  On a request with the SinkNr:

  ```
  Controller -> Slave: FBlockID.InstID.SinkInfo.Get (SinkNr)
  ```

  The following is received:

  ```
  Slave -> Controller: FBlockID.InstID.SinkInfo.Status (SinkNr, BlockWidth,
                                          ConnectionLabel, [SinkDescription])
  ```

  SinkDescription determines the kind of data that is received by the sink.
  Information about streaming data types can be found in the corresponding Stream Transmission Specification.

- **SinkName**
  Property SinkName holds the name of the streaming sink.
  It is requested with the SinkNr as parameter:

  Controller -> Slave: FBlockID.InstID.SinkName.Get (SinkNr)

  The answer is a string containing the name:

  ```
  Slave -> Controller: FBlockID.InstID.SinkName.Status (SinkNr, SinkName)
  ```

- **Connect**
  A Controller uses Connect to connect the sink to the specified connection.

  ```
  Controller -> Slave: FBlockID.InstID.Connect.StartResultAck
                                                (SenderHandle, SinkNr,
                                                 BlockWidth, ConnectionLabel)
  ```

  The sink returns as result:

  ```
  Slave -> Controller: FBlockID.InstID.Connect.ResultAck (SenderHandle, SinkNr)
  ```

- **DisConnect**
  Method DisConnect is used to disconnect a sink from the connection it is currently using.

  ```
  Controller -> Slave: FBlockID.InstID.DisConnect.StartResultAck
                                                          (SenderHandle, SinkNr)
  ```

  After the method has finished, the following is reported:

  ```
  Slave -> Controller: FBlockID.InstID.DisConnect.ResultAck
                                                          (SenderHandle, SinkNr)
  ```

- **Mute**
  The output of streaming data from a sink can be stopped by using the property Mute.

  ```
  Controller -> Slave: FBlockID.InstID.Mute.SetGet (SinkNr, Status)
  ```

  Status is On or Off to turn mute on or off.

3.2.7.2.1.3 Handling of Double Commands

Normally a repeated streaming control command (this means: Allocate / DeAllocate / Connect / DisConnect) should not occur. This handling should be done by the Connection Manager. But in an error case, the behavior of the device is defined in the following way:

Source methods

- **Allocate**
  If there is an Allocate.StartResultAck command with a source number of a currently allocated source, a normal result message with the connection label that is already assigned is sent back to the caller.
- **DeAllocate**
  If there is a DeAllocate.StartResultAck command with a source number of a currently not allocated source, a normal result message with the source number is sent back to the caller.

Sink methods

- **Connect**
  If there is a Connect.StartResultAck command with a sink number of a currently connected sink and the same connection label that it is connected to, a normal result message will be sent. If an already connected sink is to be connected to a connection with a different label, the device will first disconnect the old connection and then establish the new one. Following this, it sends out a result message.
- **DisConnect**
  If there is a DisConnect.StartResultAck command with a sink number of a currently not connected sink, a normal result message with the disconnected sink number is sent back to the caller.

3.2.7.2.2 Compensating Network Delay

The effective network delay is negligible (delay has to be smaller than 1 µs per node). No network delay compensation is required.

3.2.8 Connections

3.2.8.1 Bandwidth Management

In a MOST Network the overall available bandwidth can be divided between streaming data and packet data in a flexible way by setting the Boundary Descriptor. The Boundary Descriptor can be modified by using the mechanism provided by the MOST Network Service.

The value of the Boundary Descriptor depends on the requirements of the system and can be changed dynamically. Supervision and changing of the bandwidth or position of the Boundary is done in the TimingMaster. The TimingMaster is responsible for forwarding the information about the Boundary's position to all nodes in the network. This task is handled automatically on the MOST Network Interface Controller level.

Since the Boundary can be changed dynamically during runtime, the system can be adopted to the needs which occur in temporary or persistent situations (e.g., data download, transfer of big amounts of data like MP3, or navigation maps). In both situations a change sequence shall be performed without loss of the bus signal.
In order to adapt to the new situation, all devices must implement appropriate mechanisms. Therefore, they may have to notify themselves to the property Boundary of the TimingMaster device.

Adjusting sequence:
For managing the Boundary, the ConnectionMaster provides the method MoveBoundary.

```
Controller -> ConnectionMaster:
      ConnectionMaster.1.MoveBoundary.StartResultAck
                                        (SenderHandle, BoundaryDescriptor)
```

With the aid of this method, any initiator can request an adjustment of the Boundary. After the ConnectionMaster has received such a request, it locks itself and rejects all subsequent connection requests.
Additionally, it may inform initiators of rejected connection requests about the reason of the unavailability. The ConnectionMaster forwards the request for shifting the Boundary to the TimingMaster. This is achieved by setting the property Boundary with OPType SetGet in the device containing the TimingMaster, that is, NetBlock with InstID 0. The TimingMaster internally sets the Boundary to the requested value and re-initializes the low-level allocation mechanisms in all devices in the network. If the function Boundary.SetGet returns, the ConnectionMaster forwards the result to the initiator by the respective Result or Error message, unlocks itself, and accepts subsequent connection requests.

3.2.8.2 Streaming Connections

3.2.8.2.1 Connection Manager

Streaming connections are managed by a Connection Manager. All requests for establishing connections must be directed to this Connection Manager, which can be implemented by any device in the MOST network.
The Connection Manager is mandatory in every MOST system and represents the functions defined in FBlock ConnectionMaster (0x03); however, it is not required that the Connection Manager implements that FBlock interface and is accessible by means of FBlockID 0x03.
If the Connection Manager does provide the FBlockID 0x03 interface, that FBlock must be included in the Central Registry.

For building a point-to-point connection, FBlock ConnectionMaster provides the method *BuildConnection*.

```
Controller -> CManager: ConnectionMaster.1.BuildConnection.StartResultAck
                                        (SenderHandle, Source, Sink)
```

Source in the method above refers to any source:
Source = FBlockID.InstID.SourceNr.
Sink refers to any sink:
Sink = FBlockID.InstID.SinkNr.
CManager is the DeviceID of the Connection Manager.
After a successful connection is built, the ConnectionMaster returns:

```
CManager -> Controller: ConnectionMaster.1.BuildConnection.ResultAck
                                         (SenderHandle, Source, Sink)
```

Error handling:
If the connection fails, the ConnectionMaster answers with OPType “ErrorAck” (0x9) and the ErrorCode “ProcessingError” (0x42), and returns the parameters Source and Sink.

```
CManager -> Controller: ConnectionMaster.1.BuildConnection.ErrorAck
                                (SenderHandle, "ProcessingError", Source, Sink)
```

Removing a connection is done in an analogous way, by using the method *RemoveConnection.*

The ConnectionMaster generates an array of all existing connections including sources and sinks, where it adds more information. This array is accessible in function *ConnectionTable.*

```
Controller -> CManager: ConnectionMaster.1.ConnectionTable.Get

CManager -> Controller: ConnectionMaster.1.ConnectionTable.Status
                        (Source, Sink, BlockWidth, ConnectionLabel,
                         Source, Sink, BlockWidth, ConnectionLabel,
                         Source, Sink, BlockWidth, ConnectionLabel, ...)
```

The parameters are the same as those described above. ConnectionTable cannot be set directly. Building and removing connections is done only with methods BuildConnection and RemoveConnection.

After switching off the network, the contents of ConnectionTable are deleted, leaving no streaming connections in the system. They must be rebuilt by new requests of the initiator(s).

The ConnectionTable is deleted also when Configuration.Status(NotOK) is received by the Connection Manager since all connections are removed in this case. See section 3.1.3.2 for more information.

FUNCTIONS				
FktID	**OPType**	**Sender**	**Receiver**	**Explanation**
BuildConnection	StartResultAck	Controller	Connection Manager	Request for building connection
	ResultAck	Connection Manager	Controller	Answer with result
ConnectionTable	Get	Controller	Connection Manager	Request of that property, where the Connection Manager stores all active point-to-point connections
	Status	Connection Manager	Controller	Answer
RemoveConnection	StartResultAck	Controller	Connection Manager	Request for removing connection
	ResultAck	Connection Manager	Controller	Answer with result

Table 3-19: Functions in ConnectionMaster in conjunction with the administration of streaming connections

Deadlock prevention:
In order to prevent potential deadlocks in the connection building process, $t_{CM\_DeadlockPrev}$ is used. $t_{CM\_DeadlockPrev}$ is started as the Connection Manager makes a request to a source/sink FBlock. If the timer expires, the action will be regarded as failed.

This timer should not be used as a maximum time for the source/sink to carry out their respective operations since this must be done much faster; it is merely used to prevent deadlocks and should only be effective in the special cases where a source/sink device has malfunctioned after receiving the command from the Connection Manager.

3.2.8.2.2 Establishing Streaming Connections

When building a streaming connection, the Connection Manager uses method Allocate in the Source FBlock to connect it to the network. After a positive answer from the source, the Connection Manager uses the method Connect in the Sink FBlock to connect it to the connection used by the source. This mechanism is explained in the following figure and the text below.

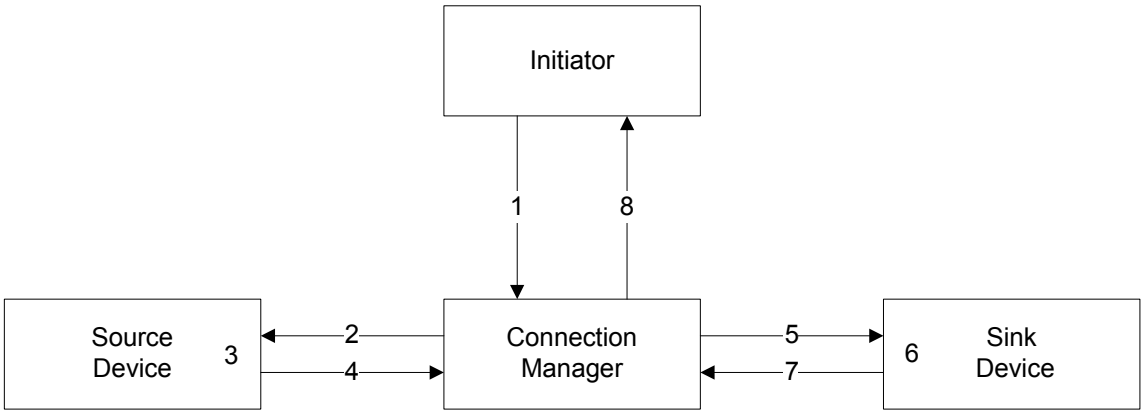

Figure 3-34: Building a streaming connection step by step

Explanation of Figure 3-34**:**

1) Method *BuildConnection* is started by an initiator, for building a connection between a source and a sink. The source must allocate its own bandwidth.

2) Connection Manager sends a command to the source device to connect its streaming output. The source must support the *Allocate* method.

3) The source tries to allocate bandwidth. The following results are possible:

   - Enough free bandwidth. Reply to Connection Manager (4) as *Allocate.ResultAck* with parameters SenderHandle, SourceNr, BlockWidth, and ConnectionLabel.

   - Not enough bandwidth. Reply to Connection Manager (4) as *Allocate.ErrorAck* with parameters SenderHandle, SourceNr, and BlockWidth.

4) The result is sent to the Connection Manager.

5) If the result is ok, the Connection Manager starts method *Connect* of the sink, communicating parameters SenderHandle, ConnectionLabel, and BlockWidth. The sink has then all the information needed of the source.

6) The Sink connects.

7) The result is sent to the Connection Manager as *Connect.ResultAck*.

8) The Connection Manager reports the result of establishing the connection to the initiator by using *BuildConnection.ResultAck*. If the building of the connection was successful, the Connection Manager internally stores the connection data. This way, if another sink is connected to this source, only the allocation data needs to be sent to the new sink.
   In case of a failure, *BuildConnection.ErrorAck* is sent and all changes to the network are undone. That means that if an error occurred in the *Connect* process, the source is disconnected and the bandwidth is freed again.

3.2.8.2.3 Removing Streaming Connections

The initiator terminates a connection previously built by the Connection Manager. The Connection Manager commands the sink to disconnect from the connection that it is currently using.
After a positive answer, and if the connection is not in use by another sink, the Connection Manager disconnects the source, using Deallocate.
After that, the Connection Manger reports the result of the termination to the initiator.

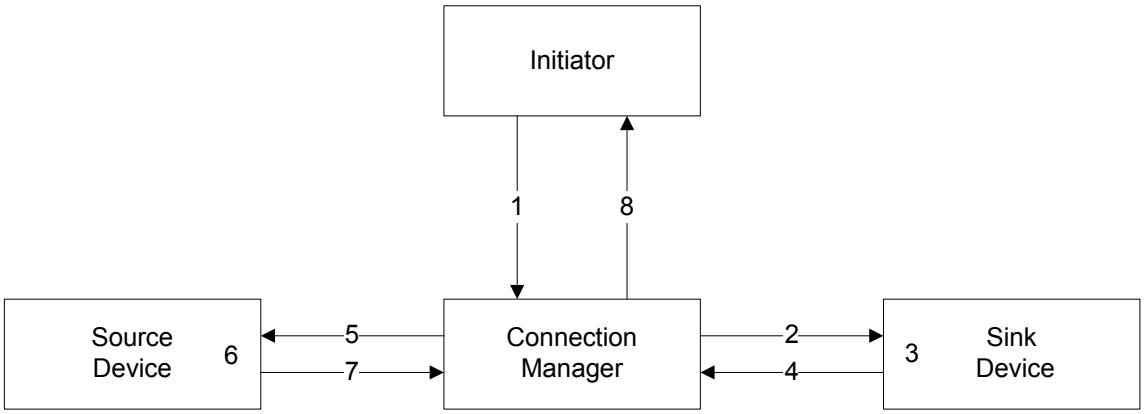

Figure 3-35: Step by step removal of a streaming connection

Explanation of Figure 3-35**:**

1) Method *RemoveConnection* is started by an initiator, for removing a connection between a source and a sink.
2) The Connection Manager starts method *DisConnect* in the sink FBlock, this makes the sink disconnect from the previously used connection.
3) The sink disconnects.
4) The sink reports the result to the Connection Manager by *DisConnect.ResultAck.*
5) If no other sink uses the connection, the bandwidth in use will be freed. Otherwise continue at step 8.
6) The source deallocates and disconnects upon reception of *DeAllocate*.
   The source has to deallocate the used bandwidth. Upon trying to do this, the following result is expected:
   - Successful de-allocation. Positive answer by *DeAllocate.ResultAck* (7).
7) The result is sent to the Connection Manager as *DeAllocate.ResultAck*.
8) If the Connection Manager has received a positive answer from the source, the connection is removed from its internal connection table. The Connection Manager sends an answer, *RemoveConnection.ResultAck,* to the Initiator. The message contains the status of the requested termination of the connection.

3.2.8.2.4 Establishing DiscreteFrame Isochronous Streaming Connections

DiscreteFrame Isochronous streaming connections (containing data and additional timebase information in parallel) can be established by the use of AllocateExt and ConnectExt. Those are identical to Allocate and Connect apart from one additional parameter: ClkSrcLabel, which is the connection label of the clock source.

3.2.8.2.5 Removing DiscreteFrame Isochronous Streaming Connections

Termination of connections is performed by the use of DeAllocate and DisConnect. However, the ConnectionMaster must not terminate the clock connection before the last data connection was terminated.

3.2.8.2.6 Supervising Streaming Connections

The availability of a connection is continuously verified by the Data Link Layer. A source malfunction leads to automatic de-allocation of allocated bandwidth.

Every streaming sink is responsible for supervising the validity of its output. If the bandwidth is de-allocated, the sink has to secure its streaming output signals and set the Mute property to on.

If the sink application is able to verify the validity of the received streaming data and the data is rendered invalid, it has to secure its streaming output signals.

When securing streaming outputs, sinks for synchronous streaming data, for example, set the output volume to zero or display a test screen.

Additional error handling can be performed by the Connection Manager (e.g., cleaning up of spontaneously vanished connections).

3.2.9 Timing Definitions

T = Timer. If expired, the implementation has to invoke the procedure as defined in the respective section of the specification (e.g., error handling).
C = Timing constraint. The implementation has to fulfill this timing requirement in order to be compliant to this specification.

Whenever a timing definition contains the character "-" instead of a value, that particular value is "not defined".

Name	Min Value	Typ Value	Max Value	Unit	Type	Definition
Initialization						
t_{Config}	2000	2000	2500	ms	T	Time that may pass after initialization of the MOST Network Interface Controller in a device until transition to NetOn state.
t_{WakeUp}	–	6	25	ms	C	Maximum time between start of activity at the Rx input of the device and start of activity at the device's Tx output. $(63 \bullet t_{WakeUp}) + t_{WaitNodes} + t_{Lock} < t_{Config}$
$t_{WaitNodes}$	–	–	100	ms	C	Time that may pass between start of activity at the Rx input of a device and the deactivation of its bypass. This timer is valid only when starting up the network.
$t_{WaitBeforeScan}$	-	-	System Integrator specific	ms	C	Time between an Init Ready event and start of a new scan by the NetworkMaster.
$t_{WaitBeforeRescan}$	-	-	System Integrator specific	ms	C	Time between broadcast of Configuration.Status(NotOK) and start of a new scan by the NetworkMaster.
$t_{DelayCfgRequest1}$	500	500	700	ms	T	Time after which the NetworkMaster starts to query nodes again that did not answer within $t_{WaitForAnswer}$. This time is used after each of the first 20 system scans since the network entered the Normal Operation state.
$t_{DelayCfgRequest2}$	10	10	11	s	T	Same as $t_{DelayCfgRequest1}$, but from the 21st attempt on.

Table 3-20: Timing definitions — Initialization

Name	Min Value	Typ Value	Max Value	Unit	Type	Definition
Shutdown						
$t_{ShutDown}$	–	2 (per node)	6 (per node)[1]	ms	C	Time between a shutdown event (i.e., stop of activity at the device's Rx input during normal operation or ring break diagnostics mode or start of activity at the input when in Slave wake-up mode) and the stop of activity at the device's Tx output. This constraint applies also as maximum response time between detection of a shutdown event, and the setting of the Shutdown Flag, in case the Shutdown Flag is not already present.
$t_{SSO\_Shutdown}$	100	100	110	ms	T	Time between setting the Shutdown Flag and switching off the signal at the output.
$t_{WaitSuspend}$	2000	–	System Integrator specific	ms	T	Time the PowerMaster waits for a ShutDown.Result(Suspend) message after broadcast of ShutDown.Start(Query).
$t_{ShutDownWait}$	1	2	15	s	T	Time the PowerMaster waits between broadcasting ShutDown.Start(Execute) and setting the Shutdown Flag.
$t_{RetryShutDown}$	9.5	10	10.5	s	T	Time the PowerMaster waits between ShutDown.Start(Query) broadcasts.
$t_{Restart}$	300	300	310	ms	T	Time after switching off the signal until the device is allowed to switch on the signal again.
$t_{PwrSwitchOffDelay}$	5	5	device specific	s	T	Time between switching off the Tx output and changing to state DevicePowerOff.
$t_{SlaveShutdown}$	16	–	System Integrator specific	s	T	Time a Slave device shall wait after ShutDown.Start(Execute). If the modulated signal was not switched off within $t_{SlaveShutdown}$, a Slave device may switch off the modulated signal.

[1] The system integrator has to ensure that the accumulated shutdown delay ($t_{ShutDown}$ of all devices) is below $t_{Restart}$.

Name	Min Value	Typ Value	Max Value	Unit	Type	Definition
$t_{WaitAfterOvertempShutDown}$	System Integrator specific	–	System Integrator specific	s	T	Time the PowerMaster waits after an overtemperature shutdown before it may restart the network.

Table 3-21: Timing definitions — Shutdown

Name	Min Value	Typ Value	Max Value	Unit	Type	Definition
General						
t_{Lock}	90	100	110	ms	T	Time a device has to see uninterrupted lock of the incoming signal before declaring Stable Lock.
t_{Unlock}	60	70	100	ms	T	Accumulated time of unlocks that lead to the detection of a Critical Unlock.
$t_{WaitAfterNCE}$	200	200	System Integrator specific	ms	T	Time between an NCE and the start of the network re-scan by the NetworkMaster.
t_{Bypass}	50	70	100	ms	T	Time the bypass of a device must stay closed after being closed. This timer is not required when starting up the network. It is only required when a node drops out of the network. $t_{Bypass} \leq t_{WaitNodes}$
t_{Answer}	–	–	50	ms	C	Time during which a NetworkSlave must respond to a query by the NetworkMaster when the system is in state NotOK.
$t_{WaitForAnswer}$	100	200	700	ms	T	Time a NetworkMaster waits for an answer from a queried Slave.
$t_{Property}$	–	–	200	ms	C	Time between complete reception of a query to a property and the start of the response message.
$t_{WaitForProperty}$ [1]	250	300	400	ms	T	Time a Shadow waits for the reception of a query to a property. The maximum value is a default value and may be adjusted by individual FBlock specifications.
$t_{ProcessingDefault1}$ [2]	System Integrator specific	100	150	ms	T	Time a device waits before sending the first Processing message. See also section 2.2.3.5.4.

[1] The maximum value is dependent on the retry settings.
[2] The System Integrator may change the default timeout value individually for every function.

Name	Min Value	Typ Value	Max Value	Unit	Type	Definition
$t_{ProcessingDefault2}$ [1]	100	100	–	ms	T	Time a device waits between sending subsequent Processing messages. See also section 2.2.3.5.4.
$t_{WaitForProcessing1}$ [1]	200	200	250	ms	T	Time a Shadow waits for the reception of the first Processing message. The timeout value has to be matched to $t_{ProcessingDefault1}$, for example, by setting it to a value 100 ms greater than $t_{ProcessingDefault1}$.
$t_{WaitForProcessing2}$ [1]	200	200	–	ms	T	Time a Shadow waits for the reception of the following Processing messages. The timeout value has to be matched to $t_{ProcessingDefault2}$, for example, by setting it to a value 100 ms greater than $t_{ProcessingDefault2}$.
$t_{CM\_DeadlockPrev}$	System Integrator specific	–	1000	ms	T	Timer to prevent deadlocks in the connection building process.
$t_{WaitForNextSegment}$	4975	5000	10015	ms	T	If the next segment of a segmented message does not arrive before this timer elapses, garbage collection is initiated.
$t_{BoundaryChange}$	100	500	5000	ms	T	Delay between sending the Boundary change notification and start of performing the actual change.

Table 3-22: Timing definitions — General

[1] The System Integrator may change the default timeout value individually for every function.

Name	Min Value	Typ Value	Max Value	Unit	Type	Definition
Ring Break Diagnosis						
t_{Diag_Signal}	1190	1200	1210	ms	T	Time a TimingSlave node waits for activity at its Rx input before switching to ring break TimingMaster mode. $t_{Diag\_Signal} > t_{Diag\_Start}$
$t_{Diag\_Master\_T1}$	2990	3000	3010	ms	T	Time the TimingMaster waits for a Stable Lock before switching to Slave mode or initiating a Diagnosis Error Shutdown.
$t_{Diag\_Master\_T2}$	490	500	510	ms	T	Time the TimingMaster stays in Slave mode checking for Stable Lock.
$t_{Diag\_Master\_T3}$	1490	1500	1510	ms	T	Time to update the node positions relative to the ring break. $t_{Diag\_Master\_T3} > t_{Diag\_Start} + t_{Lock}$
$t_{Diag\_Slave\_T1}$	2990	3000	3010	ms	T	Time the TimingSlave stays in slave mode checking for Stable Lock. $t_{Diag\_Slave\_T1} = t_{Diag\_Master\_T1}$
t_{Diag_Slave}	4990	5000	5010	ms	T	Duration of the ring break diagnosis in case of a ring break. t_{Diag_Slave} is calculated as $t_{Diag\_Master\_T1} + t_{Diag\_Master\_T2} + t_{Diag\_Master\_T3}$.
t_{Diag_Start}	–	–	1000	ms	C	Max. time between the diagnosis trigger and the actual start of the diagnosis. $t_{Diag\_Start} + t_{Diag\_Signal} < t_{Diag\_Master\_T1} - 2*t_{Restart}$ $t_{Diag\_Start} < t_{Diag\_Master\_T1} - t_{Lock}$
$t_{Diag\_Restart}$	990	1000	1010	ms	T	This is the time a device has to wait after an unsuccessful diagnosis (ring broken) until it can be restarted by network activity. $t_{Diag\_Restart} > t_{Diag\_Start} - t_{Restart}$

Table 3-23: Timing definitions — Ring Break Diagnosis

3.2.10 MOST Network Interface Controller and its Internal Services

The MOST Network Interface Controller operates the MOST bus and transmits different types of data. On top of those, higher layers are defined. The following sections give an overview of the features of the MOST Network Interface Controller that are available for the use in higher layers.

3.2.10.1 Bypass

If the bypass is closed, all signals received at the input of the MOST Network Interface Controller are forwarded to the output of the MOST Network Interface Controller. In this state, the respective device is “invisible” to the network. The device will be considered for the automatic counting of bus components only after opening the bypass, which gives access to the bus.

While the MOST Network Interface Controller is reset, the bypass is closed.

4 Appendix A: Optional OPTypes

To make a function accessible from outside, the FBlock provides a function interface (FI), which represents the interface between the function in an FBlock and its usage in another FBlock.
Function interfaces are used for debugging and testing.

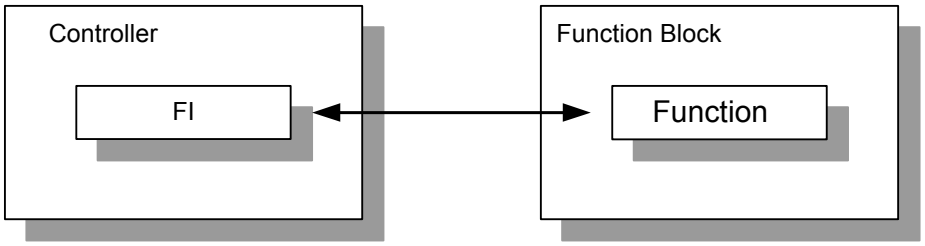

Figure A-4-1: Communication with a function via its function interface (FI)

4.1 Introduction to Function Interfaces

A function interface (FI) represents the interface between a function in an FBlock and its use in another FBlock.

To communicate with a function, a Controller or an HMI needs information about the available parameters, their limits, and the allowed operations (=FI).

In general, this information is available in the control device and is encoded in the control program. The FI was passed on, for example, like a device specification. To simplify the exchange of FIs, especially between different manufacturers, a formal description may be used that can be exchanged between the developers of Slaves and Controllers, like the well-known header files in the C programming language.

The contents of the FIs are usually known during implementation of a device (well known functions). It is also possible that FIs are transported on the bus during runtime, making it possible to dynamically reconfigure an HMI.

Example:

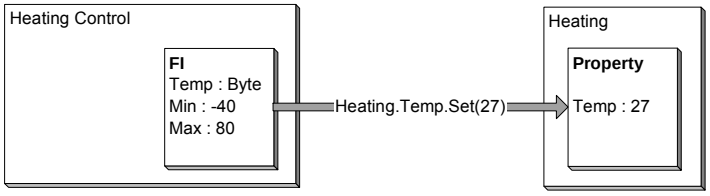

Figure A-4-2: Example of a function interface (FI)

In this example, the FI contains information about the data type of the function and about minimum and maximum value. In real implementations, an FI contains more information.

During operation, it is possible that an FI changes dynamically. In that case, all the FBlocks that have subscribed to notification will get the new interface description through the notification mechanism. For more information about notification, please refer to section 2.2.5 on page 111.

4.2 Transmitting the Function Interface

4.2.1.1 Principle

In principle, function interfaces can be transmitted to a Controller or an HMI.

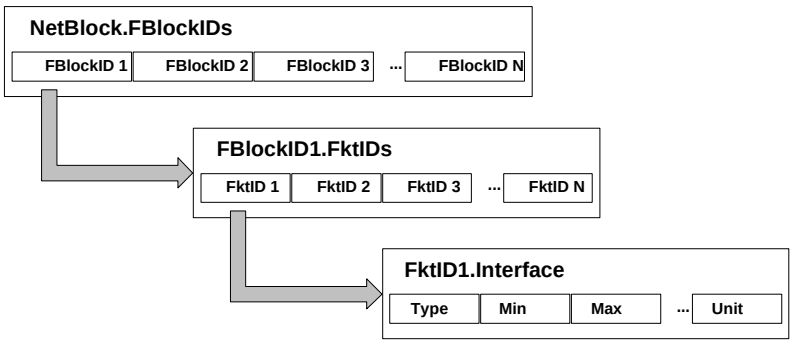

Figure A-4-3: Requesting the function interface of a function

The flow for determining all function interfaces of an FBlock looks like:

```
Controller -> Slave : FBlockID1.FktID1.GetInterface
Slave -> Controller : FBlockID1.FktID1.Interface ( [Interface Description] )
Controller -> Slave : FBlockID1.FktID2.GetInterface
Slave -> Controller : FBlockID1.FktID2.Interface ( [Interface Description] )
...
Controller -> Slave : FBlockID1.FktIDN.GetInterface
Slave -> Controller : FBlockID1.FktIDN.Interface ( [Interface Description] )
```

The parameter list “Interface Description” contains information about a function interface.

4.2.1.2 Realization of the Ability to Extract the Function Interface

In the FBlock Specifications, every interface of classified functions may be described. Thus, a classified definition of application messages, as well as a uniform description is possible, which can be based onto a few classes, described in the section 2.2.4.

5 Appendix B: Network Initialization (informative)

This Appendix contains some behavioral examples regarding Network Management. Requirements regarding Network Management behavior of the NetworkMaster and the NetworkSlaves can be found in section 3.1.2 and MOST Dynamic Specification.

5.1 NetworkMaster Section

This section contains scenarios of the behavior of the NetworkMaster during network initialization.

5.1.1 Flow of System Initialization Process by the NetworkMaster

The flow in Figure A-5-1 shows how the NetworkMaster initializes the system. Refer also to Figure A-5-2 for a flow of how the NetworkMaster performs the configuration requests from the NetworkSlaves during the system configuration.

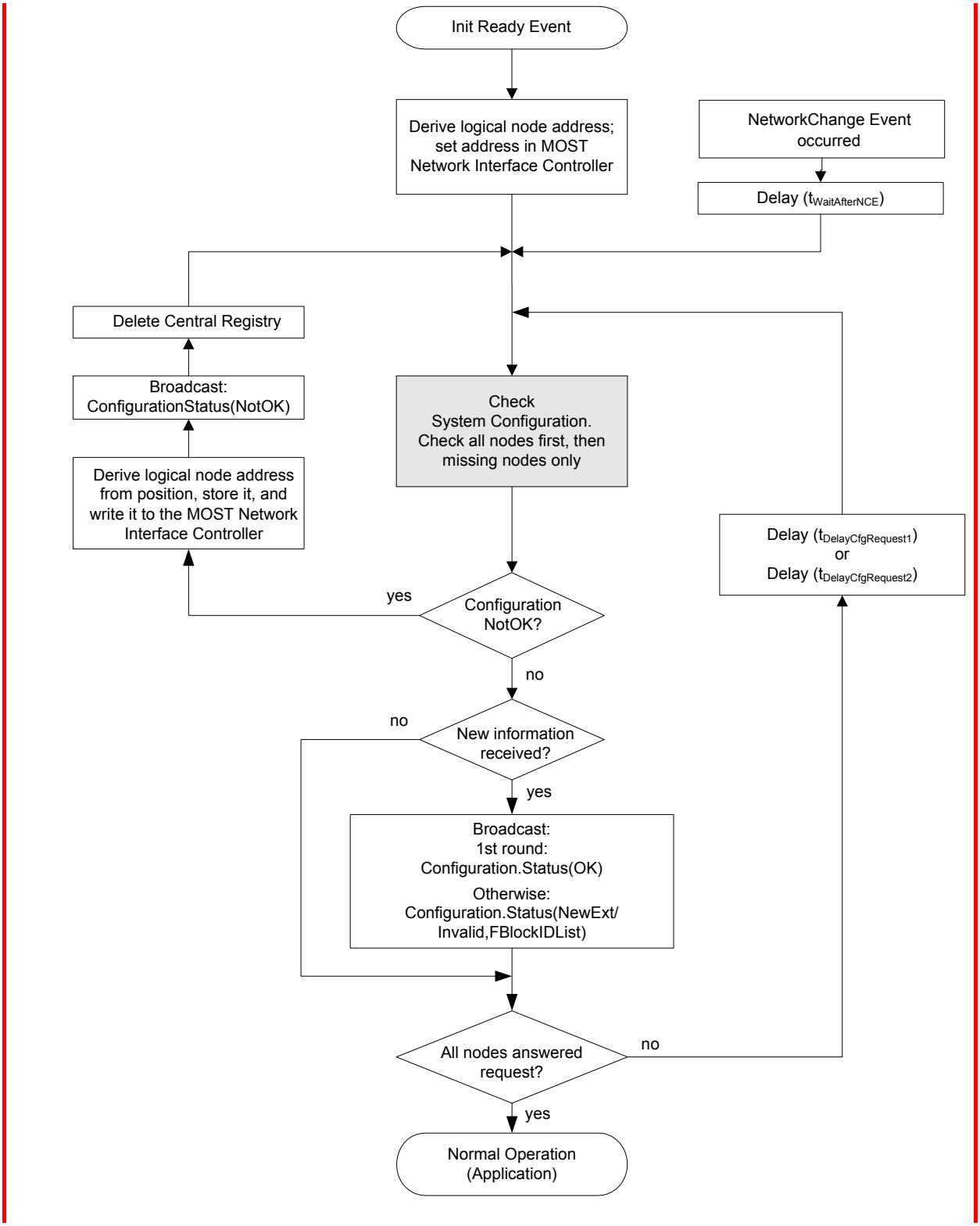

Figure A-5-1: Flow of initialization on the application level in a NetworkMaster

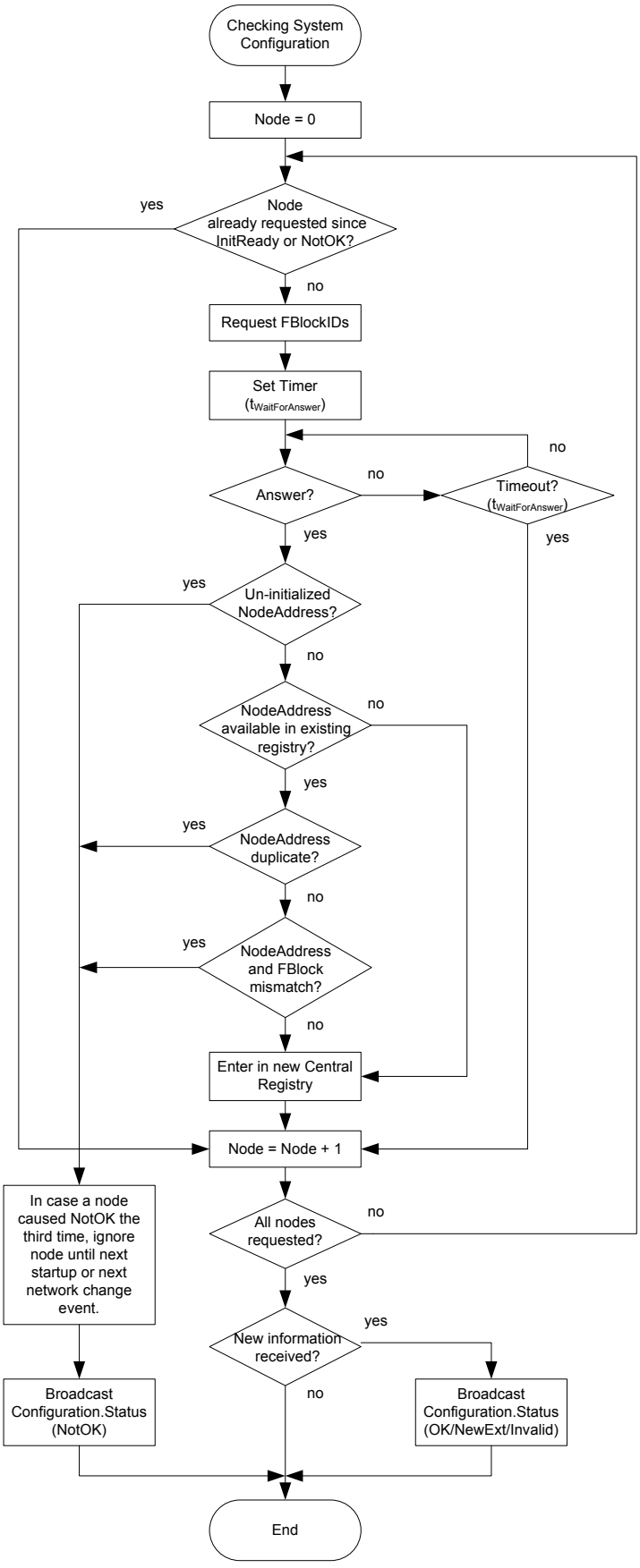

Figure A-5-2: Flow in NetworkMaster during requesting system configuration

5.2 NetworkSlave Section

The flow in Figure A-5-3 shows how a NetworkSlave behaves during System Startup and when receiving Configuration.Status messages.

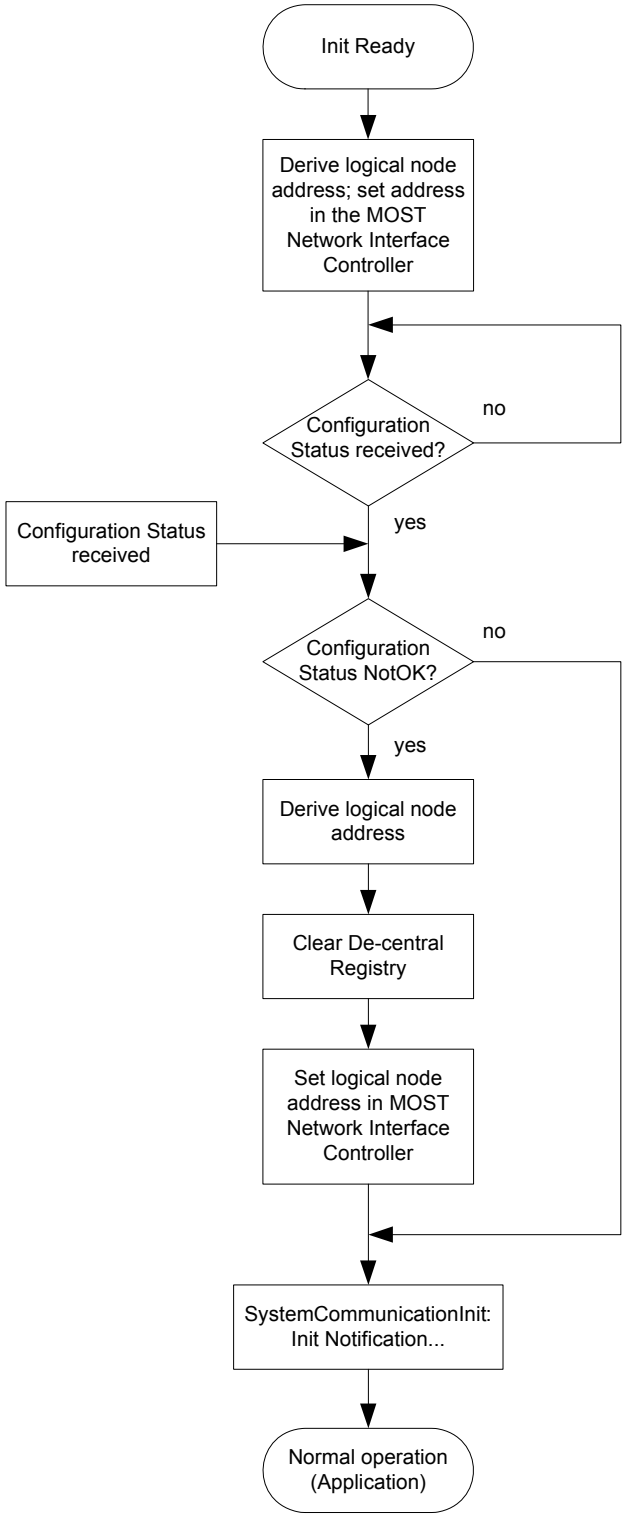

Figure A-5-3: Flow of initialization on the application level in a NetworkSlave

6 Appendix C: Typical Data Rates of Current Implementations (informative)

Speed Grade MOST150
At a system sample frequency of 48.0 kHz, the Control Channel gross data rate is 1,536 kBit/s. A control message requires at least 6 frames. This results in up to 8,000 messages per second for control messaging.

In case all messages have the maximum size of payload (TelLen equals 45), 18 frames are required and the result is 2,666 messages per second. 8,000 Control Messages per second is the best case, if none of the messages transport additional payload (TelLen is 0).

When subtracting the data used for control and data securing, the net data rate (user data plus addressing) corresponds to up to 53 bytes (45 bytes payload, 6 bytes Message ID, TelID, TelLen, and 2 bytes target address) per message:

Max: 1,130 kBit/s, in case of TelLen 45 (maximum payload, 2,666 messages)
Min: 512 kBit/s, in case of TelLen 0 (minimum payload, 8,000 messages)

A MOST device cannot use the overall available control message bandwidth of the MOST network, typically caused by limitations by the IO interface, the device driver, and the network stack.

7 Appendix D: Frame Structure and Boundary (informative)

Since the MOST system is fully synchronous, with all devices connected to the bus being synchronized, no memory buffering is needed (unlike isochronous or asynchronous devices).

The sample frequency in a MOST system can be either 44.1 kHz or 48 kHz; however, it is recommended to use 48 kHz.

7.1 Frames

The MOST frame structure is designed to allow for easy re-synchronization, as well as clock and data recovery, while guaranteeing data quality and integrity. Built-in structures provide the basis for simple network management on the lowest layers to avoid overhead.

For synchronization, two different bus node types are required. A TimingMaster that generates the frames and Slave devices that synchronize to the Master clock on the bus.

MOST150
Due to the fact that MOST150 is designed for high bandwidth, one MOST150 frame consists of 3072 bits (384 bytes). The first 12 bytes are used for administrative purposes. In this area, 4 bytes are used for control data. Therefore, a MOST Control Message gets multiplexed into 4 bytes/frame pieces. The Control Message length can vary depending on the specifics of the particular Control Message. This results in better utilization of the bandwidth regarding Control Messages. The next 372 bytes are used for packet data and streaming data transfer.

This diagram represents a MOST150 frame.

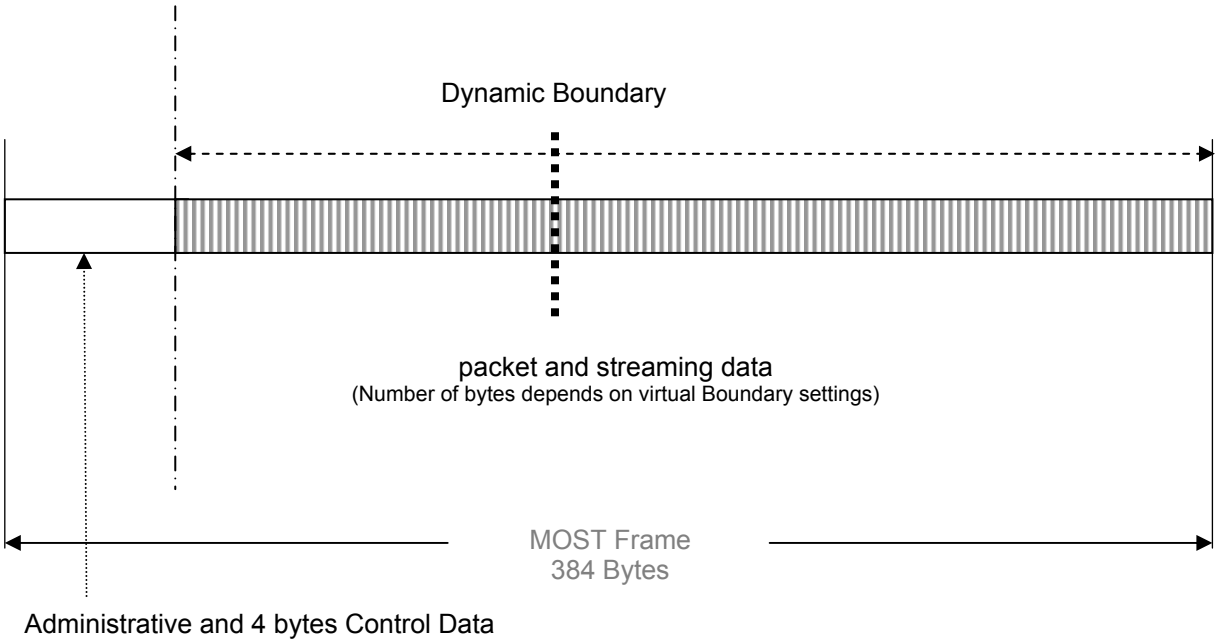

Figure A-7-1: MOST150 frame

Byte Number	Task
0 - 11	Administrative, includes ° Preamble ° System Lock Flag ° Shutdown Flag ° Boundary Descriptor ° 4 control data bytes
12 - 383	372 data bytes

Table A-7-1: Structure of the MOST150 frame

7.1.1 Preamble

The preambles are used internally to synchronize the MOST core and its internal functions to the bit stream.

For synchronization to a frame, two different mechanisms are used for TimingSlave and TimingMaster nodes. For a TimingSlave node, the first reception of valid preambles after reset, power-up, or loss of lock indicates that phase lock on the input bit stream has been accomplished.

This method ensures that the TimingSlave node is phase- and frequency-locked to the bit stream, and hence to the TimingMaster node. In the TimingMaster node, the transmitted bit stream is synchronized to an external timing source.

Once all the nodes in the network have locked to the TimingMaster's transmitted bit stream, the received bit stream has the correct frequency but will be phase shifted with respect to the transmitted bit stream. This phase shift is due to delays from each active node and additional accumulated delays due to tolerances in the phase lock within the TimingSlave nodes. The TimingMaster node re-synchronizes the received data by the use of a PLL to lock onto the incoming bit stream, thereby re-synchronizing the incoming data to the proper bit alignment.

7.1.2 MOST System Control Bits

All other bits within the frame are for management purposes on the network level. The preamble provides synchronization and clock regeneration.

7.2 Control Data Transport

Control Messages are based on a variable number of frames.

A control data message is up to 69 bytes long and has the following structure:

Area	Size / bytes	Task
Administration	12	Data Link Layer Overhead (e.g., Arbitration, Length, Transmission Status)
	2	Target address
	2	Own address (Source address)
	2	Control Channel CRC
Data	N	Data area N = 6 … L_{CMSmax}

Table A-7-2: Control data in MOST150

7.3 Boundary Descriptor

MOST systems have access to a Dynamic Boundary. This means that the bandwidth which is available for the streaming data transmission can be changed during runtime of the system, that is, during NetInterface state NetOn.

The initial value, which is used after each reset of the TimingMaster, is set in the TimingMaster's configuration. A system might not have the need to change the Boundary dynamically, but the Network Interface provides that possibility.

To change the Boundary during runtime the command NetBlock.Boundary.SetGet is sent to the NetBlock with instance ID zero (i.e., NetBlock in TimingMaster). A Slave device does not provide this property.

The only application that can allow or deny such a change is the application running on top of the TimingMaster.

Bandwidth calculation:
The total source data bandwidth is the sum of bandwidth for streaming data and packet data:
The total source data bandwidth is 117 for MOST50 and 372 for MOST150.

Total source data bandwidth = Packet bandwidth + Streaming bandwidth

The bandwidth to be used for packet transmission in number of bytes per frame is calculated according to following formula:

Packet bandwidth = maximum Packet bandwidth – (Boundary * 4)

The maximum Packet bandwidth is 116 for MOST50 and 372 for MOST150.

Property NetBlock.Boundary is designed to control the available bandwidth for the packet transmission channel (asynchronous data transmission). It is possible to use the total available bandwidth for streaming data.

NetBlock. Boundary	Available bandwidth (in number of bytes per frame)	
	Streaming data	Packet data
0	0	372
1	4	368
2	8	364
3	12	360
4	16	356
5	20	352
6	24	348
7	28	344
8	32	340
9	36	336
10	40	332
11	44	328
12	48	324
13	52	320
14	56	316
15	60	312
16	64	308
17	68	304
…	…	…
91	364	8
92	368	4
93	372	0

Table A-7-3: NetBlock.Boundary influence in a MOST150 system

NetBlock. Boundary	Available bandwidth (in number of bytes per frame)	
	Streaming data	Packet data
0	1	116
1	5	112
2	9	108
3	13	104
4	17	100
5	21	96
…	…	…
27	109	8
28	113	4
29	117	0

Table A-7-4: NetBlock.Boundary influence in a MOST50 system

8 Addendum A — MOST50 Adaption

This chapter is an addendum to the MOST Specification. It describes which requirements of the MOST Specification Rev. 3.0 E2 have to be modified to obtain compatibility of the MOST Specification and the MOST50 ePhy physical layer.

Note: *To motivate the changes, in some cases a comparison to MOST Specification Rev. 2.5 is given.*

8.1 Adaptations

The following sections provide detailed descriptions of the required adaptations to the MOST Specification Rev. 3.0 E2.

8.1.1 Application Message Service (AMS)

Related section: 3.2.5.2

8.1.1.1 Protocol Field "TelLen"

Compared to MOST Specification Rev. 2.5, the field TelLen in the AMS protocol field was increased from 4 bits to 12 bits to allow a higher application message segment size.

Available Network Interface Controllers for MOST50 use a 4 bit TelLen field on network layer.

Therefore, a dependency on the used Data Link Layer is introduced:

@MOST50: size of TelLen = 4 bits
@MOST150+[1]: size of TelLen = 12 bits

8.1.1.2 Max. Size of an Application Message Segment

In the case of MOST messages that consist of a single segment, the payload can be up to 45 bytes. For segmented messages, each segment can transport up to 44 bytes.

The maximum payload size of an application message segment is dependent on the used Data Link Layer:

@MOST50:	Single segment transfer:	up to 12 bytes
	Multi segment transfer:	up to 11 bytes

For MOST50, L_{CMSmax} is 17. $L_{AMSmax} = L_{CMSmax} - 5$ (of the L_{CMSmax} available CMS data bytes, a total of 5 bytes is reserved for Message ID, TelID, and TelLen).

Note: *For MOST50, the actual maximum payload size for groupcast or broadcast messages might be limited to 11 bytes (on transmitter side), dependent on the used Network Interface Controller.*

@MOST150+:	Single segment transfer:	up to 45 bytes
	Multi segment transfer:	up to 44 bytes

[1] MOST150 and higher speed grades.

8.1.1.3 Protocol Field “TelID”

Compared to MOST Specification Rev. 2.5, an additional identifier (0x04) has been added to the field TelID in the AMS protocol. Sending TelID 0x04 is optional; however, the receiver must be able to process such messages.

For MOST50, the receiver may ignore TelID 0x04; it must not react with a segmentation error.

8.1.2 Isochronous Data

Related section: 3.1.1.6

The availability of Isochronous Channels in a MOST50 network is optional.

8.1.3 Packet Data (16 bit addressing)

Related sections: 3.1.1.7 and 3.2.6.1.1.1

The data area size for packet data (using 16 bit addressing) was increased to 1524.

For MOST50, the data area size for packet data remains 1014. For MOST50, the maximum number of low-level retries and the time between low-level retries are fixed.

8.1.4 Tunneling Ethernet Packets

Related sections: 3.2.2.2 and 3.2.6.1.1.2

The need for the MAMAC protocol was substituted by the new 48 bit addressing mode of the Packet Data Channel.
MOST50 systems do not provide the 48 bit addressing mode for the Packet Data Channel; for MOST50 systems only the 16 bit addressing mode exists.

For MOST50, the MAMAC protocol may be used.
In this case, the TelIDs 0xA and 0xB (according to the MAMAC specification) have to be supported.

MAMAC must not be used, if the new 48bit addressing mode is available.

8.1.5 Slave Wake-Up

Related section: 3.1.2.2.2

The system can be woken by a TimingSlave device.

For MOST50, slave wake-up via MOST signal is not supported. Slave wake-up is only supported through a separate electrical wake-up line.

8.1.6 NetInterface Init

Related to the entire MOST Specification, in particular to section 3.1.2.2.2

One cause for the event Init Error Shut Down is the switching off of the modulated signal in state NetInterfaceInit.

In MOST50 systems, switching the modulated signal off is no immediate reason for an Init Error Shutdown. Therefore, Figure 3-6 is replaced by the following diagram.

Note: *In state NetInterfaceInit, the TimingMaster is allowed to set the System Lock Flag immediately on lock (i.e., it does not have to wait for stable lock).*

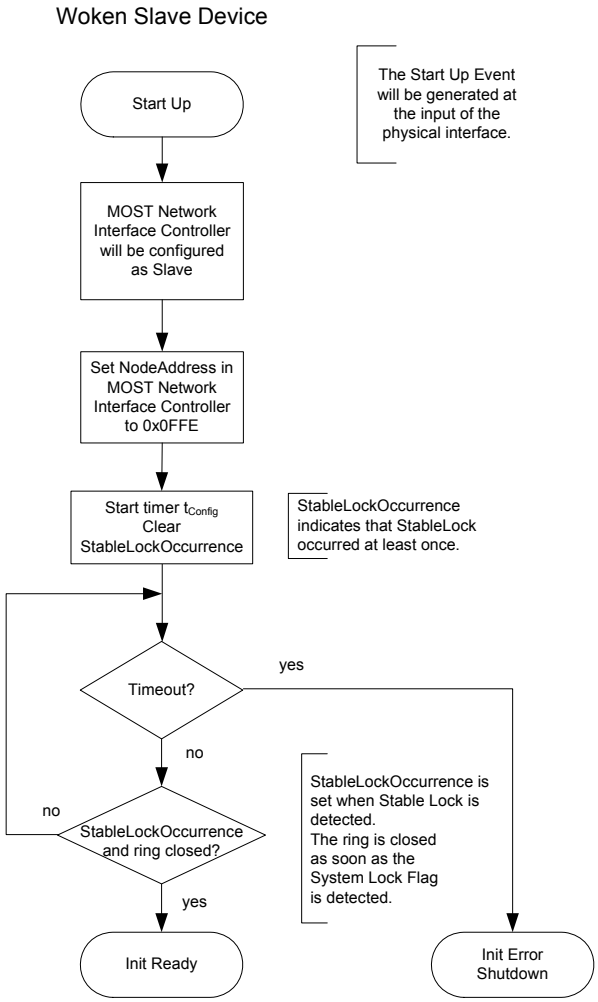

Figure 8-1: Behavior of a woken Slave device in state NetInterfaceInit

8.1.7 Bypass

Related section: 3.2.10.1

In MOST50 systems, during reset of the MOST Network Interface Controller, the output is disabled. During this time all devices located downstream detect unlocks.

8.1.8 NetInterface Normal Operation

Related to the entire MOST Specification, in particular to section 3.1.2.2.3

The MOST Specification distinguishes between Error Shutdown and Normal Shutdown. Switching the modulated signal off in state NetInterface Normal Operation results in an Error Shutdown.

In MOST50, every occurrence of an Error Shutdown event is treated as Normal Shutdown.

In MOST50 systems, switching the modulated signal off is no immediate reason for an Error Shutdown. Therefore, Figure 3-7 is replaced by the following diagram.

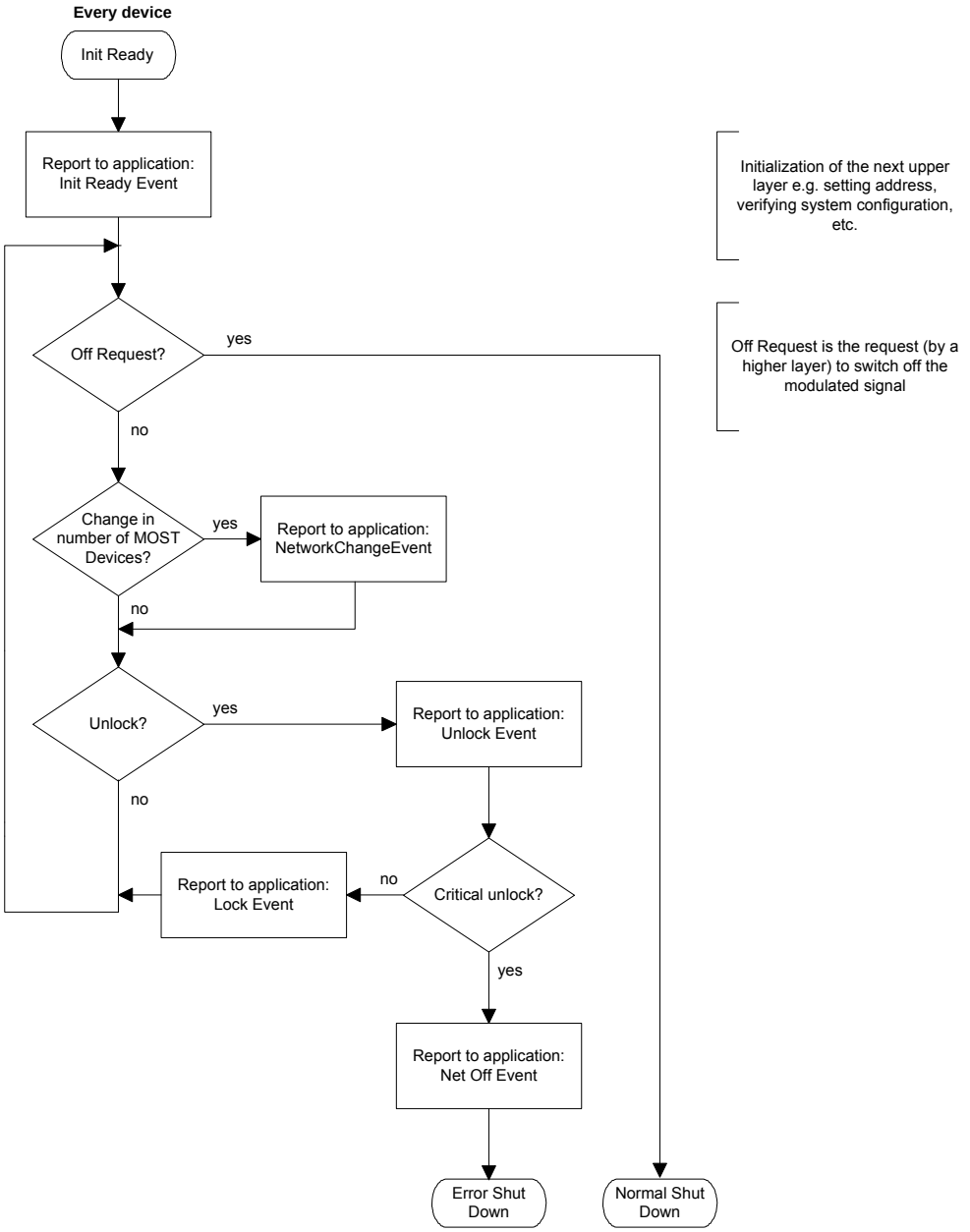

Figure 8-2: Behavior in state NetInterface Normal Operation

8.1.9 Ring Break Diagnosis

Related section: 3.1.4.1

The ring break diagnosis (RBD) feature is mandatory for all devices.

For MOST50, RBD is optional. If RBD is used, it must be implemented according to the description in section 8.1.9.1; RBD according to section 3.1.4.1 is not supported for MOST50.

The state NetInterface Diagnosis Result does not exist in MOST50 systems;

8.1.9.1 NetInterface Ring Break Diagnosis

A simple recognition of a fatal error is possible in any state. Ring break diagnosis serves the purpose of localizing a fatal error in the network.

The ring break diagnosis process can be started by various triggers, which must be chosen and implemented by the System Integrator. One possible way is to start the ring break diagnosis by disconnecting the System from the power source for a short time. In this case, ring break diagnosis is entered when signal SwitchToPower of the SwitchToPowerDetector indicates that the device was connected to power first time (e.g., after reconnection of the car's battery). After triggering ring break diagnosis all devices must start the diagnosis within t_{Diag_Start}.

In state NetInterfaceRingBreakDiagnosis, a relative node position is determined in every device. This information can be used in case of a fatal error (ring break or defective device) to localize the error.

If there is no fatal error, the NetInterface changes to state NetInterface Normal Operation.

In case of a Diagnosis Error Shut Down event, the position determined in each device describes the position relative to the device that was configured as TimingMaster at the end of ring break diagnosis (since there was no modulated signal at its input).

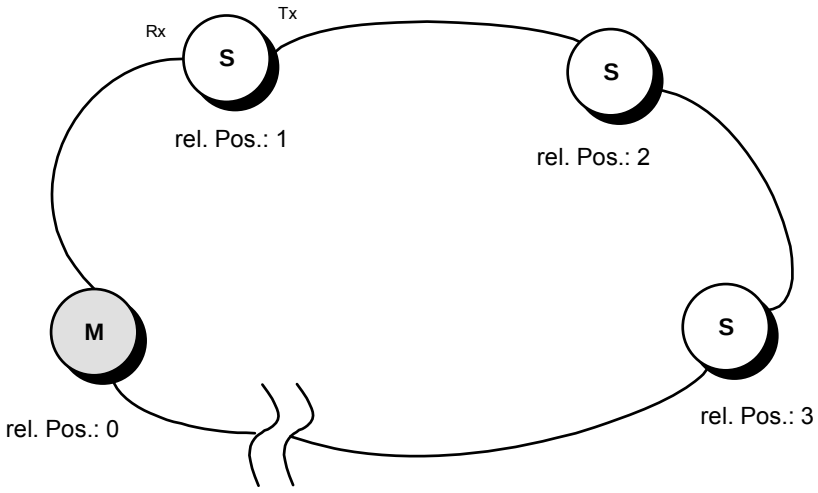

Figure 8-3: Localizing a fatal error with the help of ring break diagnosis.

Event	Transition to	Cause
Diagnosis Ready	NetInterface Normal Operation	No fatal error.
Diagnosis Error Shut Down	NetInterface Power Off	Fatal error (Ring break or defective device)

Table 8-1: Events in state NetInterface Ring Break Diagnosis

Table 8-2 lists the possible results of ring break diagnosis and the corresponding events.

Diagnosis Result	Diagnosis Info	Description	Event
No error	-	No error detected. Transition to NetInterface Normal Operation.	Diagnosis Ready
Ring Break (Position detected)	Relative position.	Ring break detected. The result indicates the relative position of ring break.	Diagnosis Error Shutdown
Diagnosis failed	-	The ring break diagnosis is inconclusive.	Diagnosis Error Shutdown

Table 8-2: Ring break diagnosis results

During ring break diagnosis, a TimingSlave device waits for a modulated signal at its input. If there is no signal after t_{Diag_Signal}, it configures itself as TimingMaster. If it recognizes a modulated signal at its input it switches back to slave mode. On a fatal error, the application stores the diagnosis result.

As soon as a device, which does not contain the designated TimingMaster, recognizes modulated signal at its input, it is configured as Slave. If no lock errors occur for a time t_{Lock} (stable lock), the relative ring position is determined.

After recognition of a stable lock and detection of a closed ring, a TimingMaster device generates a Diagnosis Ready event and changes immediately to state NetInterface Normal Operation. The Timing-Master sets the System Lock Flag in the administrative area of the MOST frame. The System Lock Flag can be read by the TimingSlave devices.

If the ring could be closed (i.e., the System Lock Flag was set), every NetInterface switches to state NetInterface Normal Operation. The application will get notified about that by the Diagnosis Ready event. After that, all high level initializations must be performed (building of the Central Registry, address initialization, notification...).

If the ring could not be closed because of a ring break, devices shall not be restarted by incoming modulated signal until $t_{Diag\_Restart}$ has passed.
The following flow charts show the behavior in the state NetInterfaceRingBreakDiagnosis.

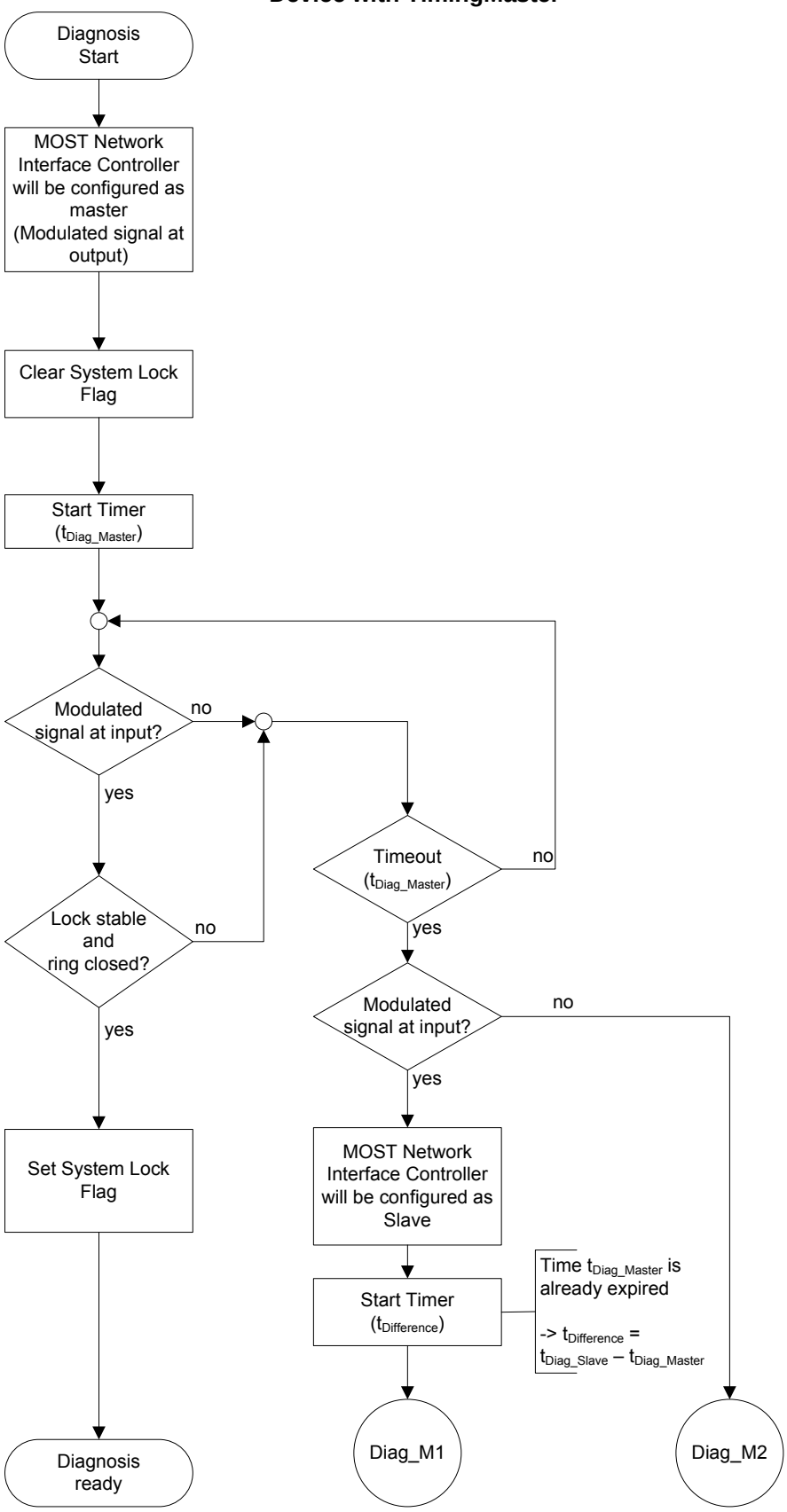

Figure 8-4: Behavior during ring break diagnosis in a TimingMaster (part 1)

Device with Timing Slave

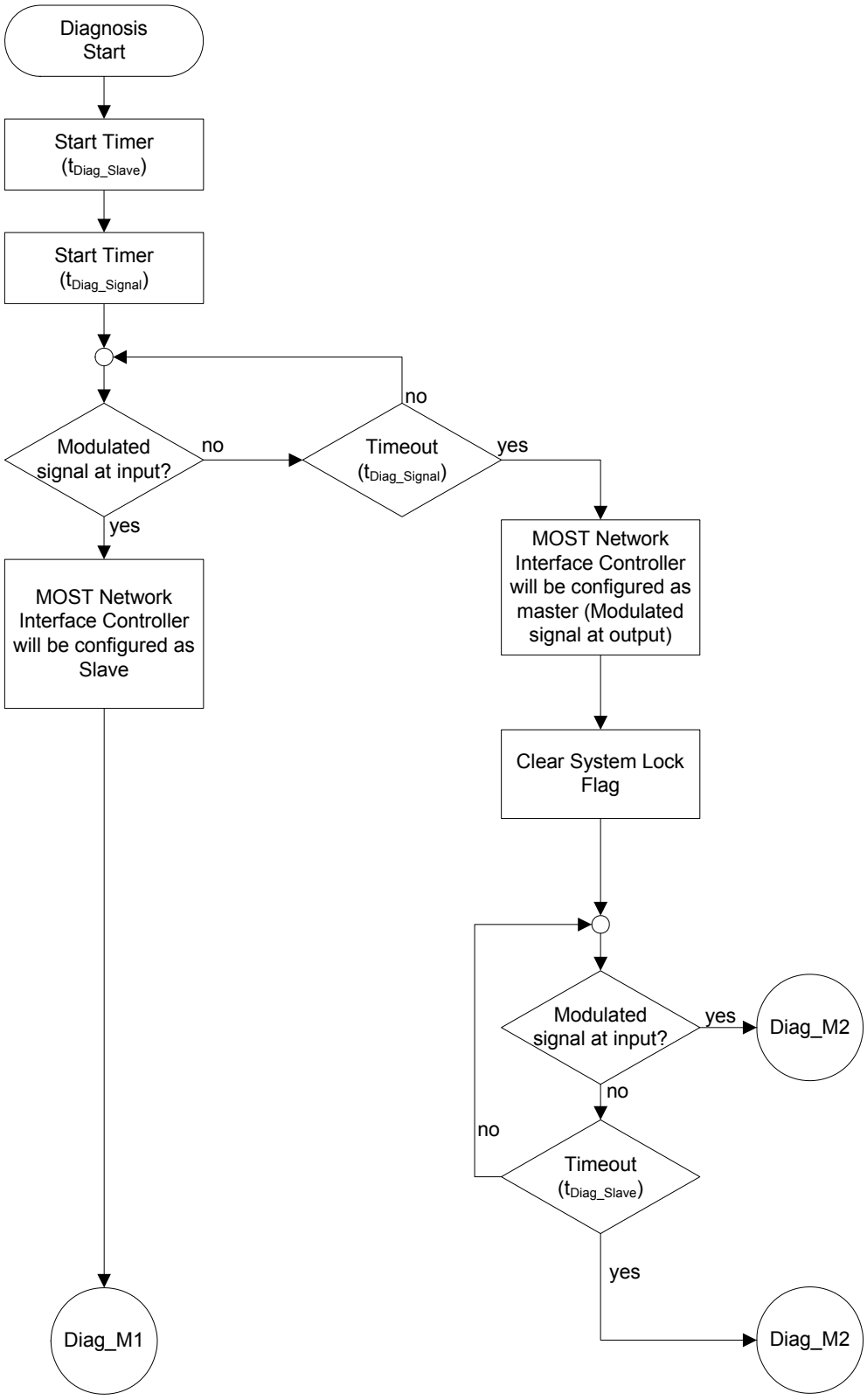

Figure 8-5: Behavior during ring break diagnosis in a Slave (part 1)

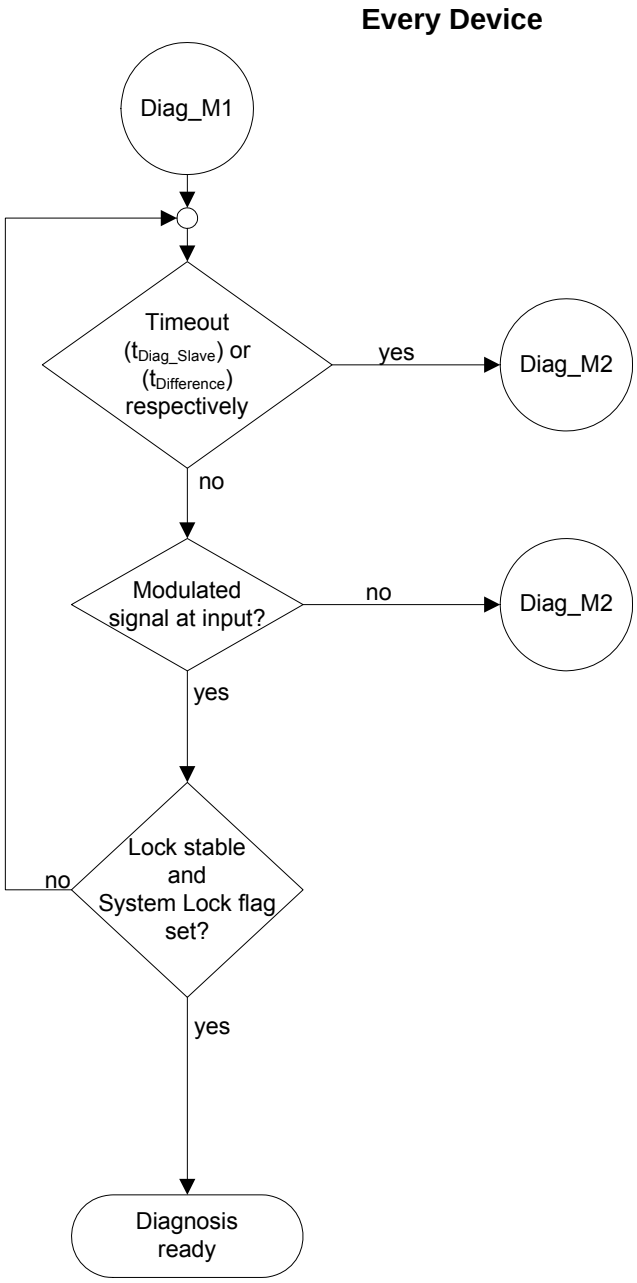

Figure 8-6: Behavior during ring break diagnosis in a TimingMaster and Slave (part 2)

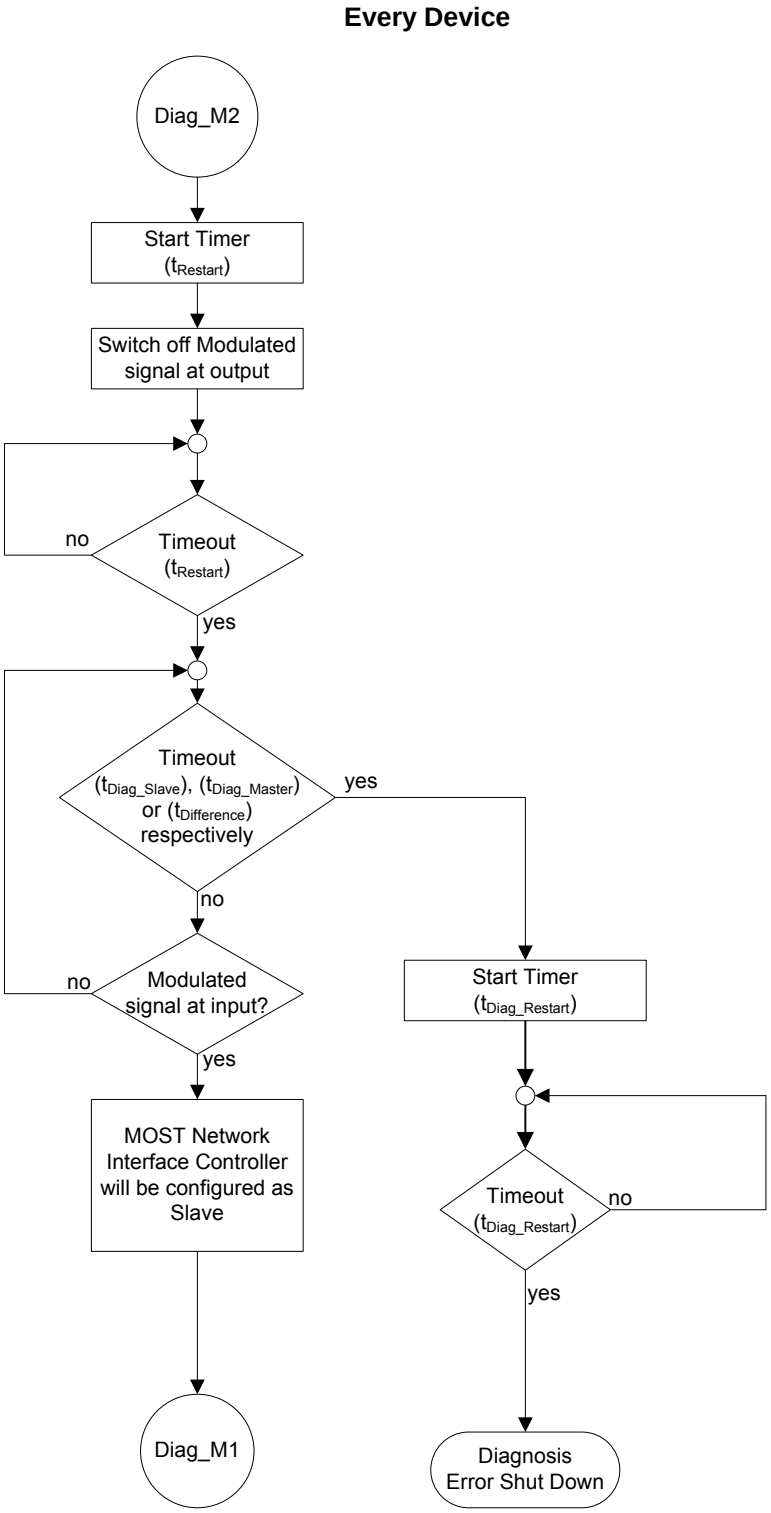

Figure 8-7: Behavior during ring break diagnosis in a TimingMaster and Slave (part 3)

Name	Min Value	Typ Value	Max Value	Unit	Type	Definition
Ring Break Diagnosis						
t_{Diag_Signal}	1190	1200	1210	ms	T	Time a TimingSlave waits for activity at its Rx input before switching to ring break master mode.
t_{Diag_Master}	2990	3000	3010	ms	T	Time the TimingMaster stays in TimingMaster mode. After this time, it may switch to TimingSlave mode.
t_{Diag_Slave}	4990	5000	5010	ms	T	Time a TimingSlave stays in ring break slave mode before generating the result of the diagnosis. In addition, $t_{Diag\_Slave} \geq t_{Diag\_Master}$ + 2s must be true.
t_{Diag_Start}	0	0	1000	ms	C	After triggering ring break diagnosis, the diagnosis has to be started within t_{Diag_Start}.
$t_{Diag\_Restart}$	990	1000	1010	ms	T	This is the time a device has to wait after an unsuccessful diagnosis (ring broken) until it can be restarted by network activity.

Table 8-3: Ring break diagnosis timing definitions

8.1.10 Shutdown and Initialization Timing Definitions

Name	Min Value	Typ Value	Max Value	Unit	Type	Definition
Shutdown						
$t_{ShutDown}$	–	2 (per node)	6 (per node)[1]	ms	C	Time between a shutdown condition and the stop of activity at the device's Tx output.
Initialization						
t_{WakeUp}[2]	–	20	30	ms	C	Maximum time between start of activity at the Rx input of the device and start of activity at the device's Tx output. $(N \bullet t_{WakeUp}) + t_{Lock} < t_{Config}$ where N is the number of TimingSlaves.
General						
t_{Bypass}	1	–	5	ms	T	When a node drops out of the network, the Network Interface Controller must stay in reset for this period of time. The lower boundary is the minimum time to detect a network change event. The upper boundary is the maximum time that does not cause a critical unlock ($t_{Bypass\_max} < t_{Unlock\_min}$).

Table 8-4: General timing definitions

[1] The System Integrator has to ensure that the accumulated shutdown delay ($t_{ShutDown}$ of all devices) is below $t_{Restart}$.
[2] The System Integrator has to ensure that the startup of the system becomes effective quicker than t_{Config}. The timing constraint t_{WakeUp} may be relaxed if the number of nodes is limited in the system.

8.1.11 Sudden Signal Off and Critical Unlock Detection

Related section: 3.1.4.2

The SSO and CU detection feature, as well as the NetBlock property NetBlock.ShutDownReason is mandatory for all devices.

This feature is speed grade dependent and not applicable for MOST50 systems.
In MOST50, there is no Shutdown Flag that would have to be set by the PowerMaster during normal shutdown.

9 List of Figures

10 List of Tables

INDEX

1

4

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Document History (Previous Revisions)

Changes MOST Specification 2V4-00 to MOST Specification 2V5-00
In general, "Section" refers to the current document, unless deletions are referred to. Sections that were deleted and not replaced by others are marked ~~strikethrough~~. If deleted sections were replaced by others, details about the substitution can be found in the change description.

Change Ref.	Section	Changes
2V5_001	General	Optical Interface changed to Physical Interface.
2V5_002	General	Removed old section 3.1.2 'Source Data Bypass' MOST25 (up to MOST Specification Rev2.4) permitted that each MOST device could be switched from active (source data bypass disabled) to passive (source data bypass enabled) to reduce the source data delay in the network. Disabling the source data bypass in a MOST25 device adds a delay of 2 network frames to streaming source data. Therefore network delay compensation is useful for noise canceling, speech recognition, and multi-channel sound applications. The bypass to reduce the network delay, as well as the delay compensation is not necessary when using MOST50. The effective delay is negligible (approx. 300 ns per node). Starting with MOST Specification Rev2.5 all devices are assumed to be "active". This section was removed because the Source Data Bypass was not used with MOST25 and it is no longer applicable for MOST50.
2V5_003	General	"light" changed to "modulated signal"
2V5_004	General	"FOT" changed to "Physical Interface"
2V5_005	General	Corrected grammar and spelling mistakes. Fixed cross-references for diagrams that were moved to other chapters (3-13, 3-14, 3-15). Unified timer names (e.g. occurrences of t_Diag_Slave were replaced by $t_{DiagSlave}$).
2V5_006	General	Unified spelling of MOST terms like NetworkMaster, ConnectionMaster, etc. Replaced "NetOn event" with "Init Ready event"; For MOST50, the "System Lock flag" was introduced.
2V5_007	General	Harmonized terminology: Source data, streaming data, packet data vs. synchronous/asynchronous data/area. Substituted "all-bypass" with "bypass".
2V5_008	Glossary	Added Glossary chapter for introduction and definition of frequently used terms.
2V5_009	1.1	Revised introduction.
2V5_010	1.4	Rephrased remark about references to improve intelligibility. Added HTTP 1.1 RFC to referenced documents.
2V5_011	1.5	Added speed grade and physical interface differentiations.
2V5_012	2.1.1.1, 2.1.1.1.2, 2.1.1.1.3, 2.1.2.3	Improved wording, eliminated redundancies.
2V5_013	2.1.4	Removed "60 bytes". Refer to the Boundary Descriptor rather than description here. Added reference to bandwidth management section.
2V5_014	2.1.2.1	Added remark about support for different physical interfaces.
2V5_015	2.2.1.1	Substituted description of "Slave, Controller, HMI" with a more precise explanation.
2V5_016	2.2.5	Rephrased paragraph on notifications to promote clarity and consistent use of terminology.
2V5_017	2.2.6	Removed statement on use of functions that did not exist during development.
2V5_018	~~2.2.9~~	Deleted sections on delegation, heredity and device hierarchy.
2V5_019	2.3.1	Removed duplicate introductory part (FBlock grouping and access) that is already contained 2.1.2.1.2 and referenced from this section.
2V5_020	2.2.3.2	Changed reserved and proprietary FBlockID ranges, added table "Responsibilities for FBlockID and FktID ranges". Added remark that Secondary Nodes apply to MOST25 only. Added note that Supplier specific FBlocks are not reported in NetBlock.FBlockIDs.Status.

Change Ref.	Section	Changes
2V5_021	2.2.3.4	Added the following: Supplier specific functions are not reported in <FBlockID>.FktIDs.Status. Notification.Set(SetAll) does not affect Supplier specific functions while on the other hand Notification.Set(ClearAll) clears Notification for Supplier specific functions. Added table “Responsibilities for FBlockID and FktID ranges”
2V5_022	2.2.3.3.6	Added description of the exceptional cases when wildcards can be used in replies. Limited “Secondary Node” error to MOST25.
2V5_023	2.2.3.5	Correction in OPType table (ErrorAck is not available for Properties and was removed).
2V5_024	2.2.3.5.1	Note added about segmented error messages. Added ErrorCode and ErrorInfo ranges for use by System Integrators and Suppliers. Added remark that “Method Aborted” must be sent, even if a method is not executed anymore. Moved example for sensor failure (Error Code 0x41) to section 2.3.12. Added remark that “Segmentation Error” might also occur in single telegram context. Added explanation of different ErrorCode and ErrorInfo positions for Error and ErrorAck. Differentiated between “Secondary Node” error for MOST25 and MOST50. Removed reference to “generally broken device” for the “Device Malfunction” error.
2V5_025	2.2.3.8.10	Added ranges for System Integrators and Suppliers in String identifier table. Added string type SHIFT_JIS (code 0x07). Reserved range therefore now starts with 0x08.
2V5_026	2.3.6.1	Substituted the abstract description of involved devices with an explanation of the setup for the example.
2V5_027	2.2.4	Entire section: Reordered OPType tables so that OPTypes appear in order of OPType ID. Removed common description (for methods and properties) from single parameter properties section (2.2.4.1) and inserted it here. Also added Trigger Method, Sequence Method, Sequence Property and Map to list of function classes.
2V5_028	2.2.4.1.1	Added footnote that parameters marked boldface are explained in detail.
2V5_029	2.2.4.1.2	Added new category “Data” to unit table. Added constants for Byte, kByte, and MByte. Renamed category “Volume” to “Miscelleaneous”. Added “%” (percent). Redefined category “Temperature” as “ Temperature and Pressure”. Added K, bar, and psi. Redefined category “Speed” as “Speed and Acceleration”. Added cm/s, °/s, and m/s^2.
2V5_030	2.2.4.1.5	Added description of DataType parameter.
2V5_031	2.2.4.1.7	Removed “Classified Stream” Parameter for Get OPType.
2V5_032	2.2.4.2.1, 2.2.4.2.2	Added function class Container to IntDesc Table. Removed reference to “internal OPTypes”.
2V5_033	2.2.4.2.3	Entire Function Class DynamicArray section exchanged.
2V5_034	2.2.4.2.4	Entire Function Class LongArray section exchanged.
2V5_035	2.2.4.2.5	Added new section: Function Class Map.
2V5_036	2.3.11.2.6	Corrected list of OPTypes in description of IntDescX.
2V5_037	2.2.4.3	Explained Sequence Method more precisely.
2V5_038	2.2.4.3.2	Added AbortAck OPType and SenderHandle parameter, which was missing for some OPTypes.
2V5_039	2.3.12	Described double registration more precisely: No second entry is generated in the Notification Matrix. Added description of behavior for invalid target addresses and effect in combination with groupcast addresses. Added list of reactions on system events. Added sensor failure example, previously under 2.3.2.5.1, and made it more generic (i.e., removed references to CAN).
2V5_040	3.2.10.1	Removed 2nd paragraph. Added bypass differentiation for oPhy and ePhy.
2V5_041	3.1.2	Removed 2nd paragraph.
2V5_042	3.1.3	Revised section.
2V5_043	3.1.3.1	Replaced section with modified MOST25 frame description and new MOST50 parts.

Change Ref.	Section	Changes
2V5_044	3.1.3.1.2	Added MOST50 Dynamic Boundary description, revised MOST25 description and added comparison table. Added note that streaming connections must be re-built after Boundary Descriptor change.
2V5_045	3.1.3.3.1	New section heading is “Control Data Transport”. Added minimum/maximum number of frames for MOST50.
2V5_046	3.1.3.3.4	Renamed section to “Distinction between Source Data and Control Data”.
2V5_047	3.1.3.3.5	Rephrased to match both MOST25 and MOST50.
2V5_048	3.1.3.3.6	Changed heading from “Synchronous Area” to “Streaming Data”. Added MOST50 description.
2V5_049	3.1.3.4.2	Changed heading from “Asynchronous (Packet Data) Area” to “Packet Data”. Added note on limited applicability for MOST50.
2V5_050	3.1.3.4.2	Restructured and separated MOST25 part from general description. Added paragraph on MOST50 frame rate and message rate.
2V5_051	3.1.4.1	Added address range 0x1000...0xFFFE “Reserved for future use” Changed range 0x440...0x4FF to “Reserved”. Removed last row in Address range table (“highest nibble reserved for future use”). Substituted “Unique node address” with “Logical node address” for consistency. Added distinction between dynamic and static logical node addresses.
2V5_052	3.1.4.3	New section ‘Automatic Allocation Mechanism’. Merged former sections 3.1.5.4 ‘Automatic Channel Allocation’ and 3.1.5.5 ‘Detection of Unused Channels’ into this section. Added MOST50 description, separated from MOST25 part.
2V5_053	3.2.1	Added description of Primary and Secondary Nodes concept. Removed names of application states; they are not relevant to the specification.
2V5_054	3.1.2.2.2	Revised the 2nd bullet. Included more general description of Init Ready event and System Lock flag.
2V5_055	3.1.2.2.2	Changed diagrams so that MOST25 and MOST50 are covered.
2V5_056	3.1.2.2.3 3.1.5.2	Moved and revised parts of sections ‘NetInterface Normal Operation’ and ‘Unlock’.
2V5_057	3.1.2.2.4	Added ring break diagnosis results table. Separated paragraph with applicability limited to MOST25. Improved diagrams and distinguished between MOST25 and MOST50. Removed remark on evaluation of signal during NetOn.NetInterfaceNormalOperation.
2V5_058	3.2.3	Merged former sections 3.10 and 3.3.3.2.6 ‘Secondary Nodes’ into this section. Tagged entire section as “MOST25 only” because MOST50 does not support Secondary Nodes.
2V5_059	3.1.2.3.1	Replaced AbilityToWake by PermissionToWake and CapabilityToWake.
2V5_060	3.1.5	Revised whole section ‘Error Management’.
2V5_061	3.1.5.5	Changed heading from ‘Low Voltage’ to ‘Supply Voltage’
2V5_062	3.1.5.6	Replaced “This recommendation is applicable…” with “This section applies…”
2V5_063	3.2.5.6.2	Mentioned $t_{WaitAfterOvertempShutDown}$ as relevant timer for restart attempts. Added note that the PowerMaster must support either re-start by its own decision or by user request.
2V5_064	3.1.3.1.1.1, 3.1.3.3.2.3, 3.1.3.3.3, 3.1.3.3.3.2	Removed parts that refer to a stored Central Registry.
2V5_065	3.1.3.1.2.3	Deleted parts of the section (device failure handling).
2V5_066	3.1.3.2	Added Shutdown.Start(Execute) transition from System State OK to NotOK. Added footnote stating that logical node addresses will not be recalculated for that transition.

Change Ref.	Section	Changes
2V5_067	3.1.3.3.3.2	Content of “Stable Network” section included here because it is was not relevant to 3.1.3.3.3.1, where a similar description is already referenced.
2V5_068	3.1.3.3.5.4	Clarification: New InstID is set by using the logical node address.
2V5_069	3.1.3.3.6.1	Added remark that Configuration.Status(NotOk) must be sent after ConnectionMaster becomes unregistered.
2V5_070	3.3.3.6.3	Added remark that Configuration.Status(NewExt) with an empty list can only be sent after completion of the scan.
2V5_071	~~3.3.3.8~~	Removed “Verification Scan” and “Missing Devices” sections because the concept of a stored Central Registry was abandoned.
2V5_072	3.1.3.4.1.3	Changed description of conditions under which the Decentral Registry is cleared.
2V5_073	~~3.3.4.1.4~~	Removed “Persistence of the Decentral Registry” section.
2V5_074	3.1.3.4.3.3	Added recommendation not to include FBlocks with InstIDs derived from position in NetBlock.FlockIDs.Status or Central Registry.
2V5_075	3.1.3.4.3.7	Added note stating that the device also assumes System State NotOk when the NetInterface enters normal operation.
2V5_076	3.1.3.4.3.8	Removed Configuration.Status(Invalid, <empty>) from the list of examples.
2V5_077	3.1.3.4.3.15	Added description of reinitialization behavior regarding streaming/packet connections and notifications.
2V5_078	3.2.2	Emphasized that the example where address ranges are assigned according to device functionality relies on the static address range. Added remark that group address can be stored optionally. Improved the description of group address handling in case of device power loss.
2V5_079	3.2.3	Removed reference to “Register” to make the description more generic.
2V5_080	3.4.4	Removed section on High Level Retries. These are implemented according to the System Integrator’s own policy. As a result of the deletion, previous section 3.4.5 “Basics for Automatic Adding of Device Address” has become 3.4.4. Heading numbers of the following level 3 headings under 3.4 have decreased by one.
2V5_081	3.2.5.2	Added remark that telegram length has to be exact.
2V5_082	3.2.7.1	Revised section to a more general description regarding data and channels.
2V5_083	3.2.7.2.1.1, 3.2.7.2.1.3	Revised to better differentiate between MOST25 and MOST50.
2V5_084	3.5.2.2.4	New section “Order of Streaming Channel Lists” created.
2V5_085	3.2.7.2.2	Revised section.
2V5_086	3.2.7.2.1.1	Added footnote about SourceConnect. Added footnote that SourceActivity is optional. Revised section to a more general description.
2V5_087	3.2.6	More generalized description for MOST 25 priorities, previously under 3.6.1.1.
2V5_088	3.2.6.1	Removed previous section 3.6.1 on “Direct Access to the MOST Network Interface Controller”. Revised section (previously 3.6.2) to a more general description regarding different types of telegrams.
2V5_089	3.6.1.1	Added examples for different telegram types.
2V5_090	3.2.8	Former section 3.7 Managing Synchronous/Asynchronous Data removed. The content was already covered under the ConnectionMaster section. Previous section 3.8 “Connections” now resides under 3.7.
2V5_091	3.2.8.1	New section “Bandwidth Management”, containing description of Boundary Descriptor adjustment.
2V5_092	3.2.8.2.2	Specified that a source shall perform 20 retries before Allocate is considered failed by the ConnectionMaster.
2V5_093	3.7.2.4.2	Slightly restructured and rephrased.
2V5_094	3.2.8.2.1	Revised.

Change Ref.	Section	Changes
2V5_095	3.2.9	Updated description of timer t_{Bypass}. Changed description of $t_{SlaveShutdown}$ to improve intelligibility. Revised description of $t_{DelayCfgRequest1}$ and $t_{SlaveShutdown}$. Added timer $t_{WaitAfterOvertempShutDown}$, $t_{WaitForNextSegment}$ and $t_{MsgResponse}$.. Renamed t_{Diag_Light} to t_{Diag_Signal}. Added $t_{WaitForProperty}$ note on overwriting default maximum value by FBlocks.
2V5_096	3.2.10	Replaced entire chapter with content from WG Phys. Layer.
2V5_097	5.1.1	Previous section 5.1.1 “System Startup when a Central Registry is Available” removed because the concept of a stored Central Registry has been abandoned. Section 5.1.1 now contains “Flow of System Initialization Process by the NetworkMaster”. Replaced “old registry” with “existing registry” in Figure A-5-2.
2V5_098	Appendix B	Removed previous Appendix B “Synchronous Data Types”. The contained information is now part of the “MOST Multimedia Streaming Specification”. Examples for typical data rates for MOST25 and MOST50 are now contained in Appendix B.
2V5_099	Document History	Added remark to emphasize that section references are based on the current document. Moved document history for versions prior to 2.5 to the end of the document.

Changes MOST Specification 2V3-00 to MOST Specification 2V4-00

Change Ref.	Section	Changes
2V4_001	General	Old chapter 4.2.2 deleted.
2V4_002	General	Old chapter 4.2.3 deleted.
2V4_003	2.1.1.1.1	Removed sentence about asynchronous channel administration.
2V4_004	2.1.2.1	Changed description for CD player.
2V4_005	2.2.8	System Service changed to Network Service.
2V4_006	2.2.3.5	Complemented table.
2V4_007	2.2.3.5.1	New wording for Segmentation error with Error Info 0x04.
2V4_008	2.2.3.5.1	Clarified ErrorCode 0x01.
2V4_009	2.2.3.5.1	Reserved error code for supplier specific error codes.
2V4_010	2.2.3.5.1	Improved language
2V4_011	2.2.3.5.1	For clarification, ErrorCode 0x02 is also explained
2V4_012	2.2.3.5.1	Error codes removed 0x08 and 0x09 removed from paragraph “Application error – Parameter error”.
2V4_013	2.2.3.5.1	Added examples to ‘Syntax error’ and ‘Error secondary node’.
2V4_014	2.2.3.5.1	Updated figure 2-15.
2V4_015	2.2.3.5.5	Added timer $t_{WaitForProperty}$.
2V4_016	2.2.3.5.7	Added timer $t_{WaitForProperty}$.
2V4_017	2.2.3.5.10 2.2.3.5.11	Changed description for Abort and AbortAck.
2V4_018	2.2.3.6 2.2.3.8.13	Added data type short stream.
2V4_019	2.2.3.8.2	Representation of Boolean Data Types
2V4_020	0	Clarification that there is no coding Byte before the string and the string is always coded in ASCII.
2V4_021	2.2.1 2.2.2	InstID changed from 0 to 1.
2V4_022	2.3.11.1	Added OPTypes SetGet and Get to function classes Switch and Number in table.

Change Ref.	Section	Changes
2V4_023	2.2.4.2.1 2.2.4.2.2	Updated tables that describe IntDesc.
2V4_024	2.2.4.2.3	Deleted EI4 in example.
2V4_025	2.2.4.2.4	Completed tables with lost data.
2V4_026	2.2.4.2.4.2	Changed wording for ArrayWindow.
2V4_027	2.2.4.3.2	Updated parameter list for Interface in table.
2V4_028	2.3.12	Increased description of deletion of entries.
2V4_029	2.3.12	Notification changed to FktID.
2V4_030	3.2.10.1	Section name Electrical Bypass changed to only Bypass.
2V4_031	3.2.10.1	Changed description for electrical bypass.
2V4_032	3.1.4.1	Reserved device address 0x0FF0 as optional for debug purpose.
2V4_033	3.1.3.3.4 3.1.2.2.2 3.1.2.2.3 3.1.2.2.4 3.2.6 6	Removal of MOST transceiver register references.
2V4_034	3.1.2.2.2 3.1.5.1.2	Replaced t_{Master} and t_{Slave} with t_{Config}.
2V4_035	3.1.2.2.4	Replaced t_off by t_restart in Figure 3-14.
2V4_036	3.1.2.3	Changed description.
2V4_037	3.1.2.3.2 3.2.9	Added timer $t_{SlaveShutdown}$.
2V4_038	3.1.2.3.3.1	Increased description for “Request Stage”.
2V4_039	3.1.5.3	New definition of NCE.
2V4_040	3.1.3.2.2	Wording changed in bullet number 4.
2V4_041	3.1.3.2.2	The Connection Manager must not de-allocate channels.
2V4_042	3.1.3.3.1.2 3.3.3.3.4 3.2.9	Changed timer $t_{WaitAfterNetOn}$ to $t_{WaitBeforeScan}$ and extended timer to cover Configuration.Status(NotOk).
2V4_043	3.1.3.3.4.7	Changed description.
2V4_044	3.1.3.3.6.3	Changed the headline.
2V4_045	3.1.3.3.6.3	NWM should send a Config.New with an empty list when a network scan that was triggered by an NCE could not detect any changes to the registry.
2V4_046	3.1.3.4.3.13	Chapter updated.
2V4_047	3.2.2	Group Address part extended.
2V4_048	3.2.2	Four changed to five.
2V4_049	3.2.3	Reduced chapter about control message priority.
2V4_050	3.5.2.1	Improved language.
2V4_051	3.2.7.2.1	Channel lists must always be in ascending order.
2V4_052	3.2.7.2.1.1	Increased requirements for Connection Manager.
2V4_053	3.2.7.2.1.3	Sink changed to source.
2V4_054	3.7	Deleted part that describes Boundary.
2V4_055	3.2.8.2.1	InstID changed to 1.
2V4_056	3.2.8.2.1	Extended parameter lists.
2V4_057	3.2.8.2.2	ResultAck changed to Result.
2V4_058	3.2.9	Added new timer $t_{WaitForProperty}$.

Change Ref.	Section	Changes
2V4_059	3.10	A third scenario added, where primary node handles Ctrl + Stream and the secondary node handles Packet.
2V4_060	3.10.2	Extended for clarification.

Changes MOST Specification 2V2-00 to MOST Specification 2V3-00

Change Ref.	Section	Changes
2V3_001	General	NetServices replaced by Network Service
2V3_002	General	MOST Transceiver replaced by MOST Network Interface Controller
2V3_003	General	Function Catalog replaced by FBlock Specification
2V3_004	General	Differentiation between all-bypass and source data bypass.
2V3_005	General	Old chapter 3.1.5.2 removed.
2V3_006	General	Old chapter 3.1.5.7 removed
2V3_007	General	Old chapter 3.3 moved to 3.4 and some contents distributed to other chapters.
2V3_008	General	3.3.8 “Direct Access to OS8104” and 3.3.9 “Remote Control” were removed.
2V3_009	1	Chapter reworked, new document structure.
2V3_010	2.1.2	Connection Manager introduced.
2V3_011	2.1.4	MOSTSet Boundary removed
2V3_012	2.2.8	Is now MOST Network Service overview. Picture changed.
2V3_013	2.3.2.2	Connection Master not mandatory. FBlock Enhanced Testability added.
2V3_014	2.3.2.3	Description of InstIDs improved. Section on handling InstID of FBlock Enhanced Testability added.
2V3 015	2.3.2.3.2	Added Note: Wildcard must not be used for InstID assignment.
2V3 016	2.3.2.3.4	InstID NWM added
2V3_017	2.3.2.5.1	Error codes 0x08 and 0x09 deprecated. Application notified when segmentation error occurs. No ErrorAck on segmentation errors.
2V3_018	2.3.2.5.3	Added t_ProcessingDefault1, t_ProcessingDefault2, t_WaitForProcessing1, t_WaitForProcessing2 to text and pictures.
2V3_019	2.3.2.5.5	$t_{Property}$ introduced.
2V3_020	2.3.2.5.7	$t_{Property}$ introduced.
2V3_021	2.3.2.7	Reference to MOST High removed.
2V3_022	2.3.2.7.10	DAB Charsets added.
2V3_023	2.3.8	FBlock Enhanced Testability does not need to be listed.
2V3_024	2.3.11.2	Overview table added.
2V3_025	2.3.11.2.4.2	Reference to Layer 2 of NetService removed.
2V3_026	2.3.11.2.5	Function Class Sequence Property added.
2V3_027	2.3.11.3	Overview table added.
2V3_028	2.3.11.3.2	Function Class Sequence Method added.
2V3_029	2.3.12	Error handling extended.
2V3_030	3.1.1	Changed “Rx pin” to “Tx pin”
2V3_031	3.1.4.4.2	Remote Access removed. Transceiver register reference removed. Standalone mode removed. Remote GetSource added.
2V3_032	3.1.5.1	SAI removed. Table changed.
2V3_033	3.1.5.2	Standalone mode removed.
2V3_034	3.1.5.3	Transceiver specifics removed
2V3_035	3.1.5.4	Transceiver specifics removed. Time estimation removed.
2V3_036	3.1.5.8	Rewritten without transceiver registers.
2V3_037	3.2.2.2	As soon as the initialization of the MOST Network Interface Controller starts, the logical node address in the MOST Network Interface Controller has to be set to 0x0FFE.
2V3_038	3.2.2	Setting logical node address in MOST Network Interface Controller has been added to Figure 3-4, Figure 3-5, and Figure 3-6. Note that Figure 3-5 and have switched places from previous version.
2V3_039	3.2.2.4	t_Diag_Start and t_Diag_Restart added
2V3_040	3.2.2.4	Figure 3-10 and Figure 3-12, Diagnosis Normal Shut Down replaced by Diagnosis Ready.
2V3_041	3.2.4	Power Management section reworked. The Shutdown procedure has been divided into Network Shutdown and Device Shutdown. The Device Shutdown procedure is new.
2V3_042	3.2.5	Configuration.Status NotOk added as a case for securing synchronous data. Also changed so that sinks have to mute in case of an error. Not sources.

Change Ref.	Section	Changes
2V3_043	3.2.5.4	New definition of Network Change Event.
2V3_044	3.2.5.5	Note rewritten and maximum changed to 11 Bytes. NWM has InstID changed to 1
2V3_045	3.2.5.7	Failure of a Network Slave Device added.
2V3_046	3.2.5.8	Figure 3-17, Application may PowerOff in “Device Standby”. Voltage levels are no longer exact. Application removed from power states. Note added.
2V3_047	3.3	Network Management section is new. This section replaces section 3.2.3 and 3.3.5 of MOST Specification V2.2.
2V3 048	3.3.3.3.4	Introduced a new timer $t_{WaitAfterNetOn}$.
2V3 049	3.3.4.3.2	Deleted : The exception…(rest of paragraph)
2V3 050	3.3.4.3.8	Determination of System State clarified
2V3 051	3.4	This is old chapter 3.3
2V3 052	3.4.1	Address descriptions changed and Internal Node Communication Address added.
2V3 053	3.4.5	Section contains an example of Basics For Automatic Adding of Physical Address. This section is compiled from parts of section 3.3.5 of MOST Specification V2.2.
2V3 054	3.5	Chapter reworked and SourceConnect added. Mute was changed to SetGet.
2V3 055	3.5.1	NetServices routines removed.
2V3 056	3.5.2.1	Added remark that the SourceHandles function should only be used for debugging purposes.
2V3 057	3.5.2.2	Added that sources and sinks are numbered in ascending order starting from 1.
2V3 058	3.6.2.1	Ethernet Frames replaced by MAMAC Packets.
2V3 059	3.7	Boundary is now SetGet and NWM has InstID 1 in example.
2V3 060	3.8	Reworked. t_DeadlockPrev added. t_CleanChannels added and RemoteGetSource mentioned in Supervising Synchronous Connections.
2V3 061	3.8.1.4	Reworked supervising synchronous connections
2V3 062	3.9	New Timing Definition table.
2V3 063	4.1	Picture is only an example solution.
2V3 064	4.2.2	Table removed.
2V3 065	4.3	Standalone mode removed.
2V3 066	4.6	Absolute power values removed from the picture. Relative values introduced. SwitchToPower detector is optional.
2V3 067	4.7	Absolute power values replaced by relative values. Application changed to device.
2V3 068	Appendix A: Network Initialization	Added. Contains information from old chapter 3.4.

Changes MOST Specification 2V1-00 to MOST Specification 2V2-00

Change Ref.	Section	Changes
2V2_001	Bibliography	- Added [9] MAMAC Specification.
2V2_002	1	- Figure 1-1 updated with MAMAC. Function catalog has been split
2V2_003	2. 2. 2	- Events are only generated if requested.
2V2_004	2. 2. 4. 2	- SetResult changed to SetGet. Incrementing/decrementing added to the table.
2V2_005	2. 2. 10. 1	- Removed a table and the surrounding text.
2V2_006	2. 3. 2. 2	- Added Mandatory to Table2-2. Added the following function blocks to Table 2-2 and Table 2-3: Handsfree Processor: 0x28 DVD Video Player: 0x34 TMC Decoder: 0x53 Bluetooth: 0x54
2V2_007	2.3.2.3	- Changed default instance ID to 0x01. Removed two sentences that had to do with the old way the instance ids worked.
2V2_008	2.3.2.4	- Added NotificationCheck.
2V2_009	2.3.2.5.1	- Re-numbered the list correctly. Added text to Method Aborted
2V2_010	2.3.2.5.2	- Removed Result and Processing from the headline. Rephrased the text.
2V2_011	2.3.2.5.3	- Figure 2-16 and Figure 2-17 now use “yes” and “no”.
2V2_012	2.3.2.5.7	- Added information about the Get part of SetGet.
2V2_013	2.3.2.5.9	- Added Status, Error to the headline.
2V2_014	2.3.2.5.10 2.3.2.5.11	- Added Error/ErrorAck to the headlines. - Added references to Method Aborted.

Change Ref.	Section	Changes
2V2_015	2.3.2.7	- Classified stream added. Also a note about MOST High was added.
2V2_016	2.3.2.7	- Values of example 2 corrected. The first sentence on the same page was re-written.
2V2_017	2.3.2.7.10	- RDS character set added and a warning about RDS strings size. - Also added warning that character sets can't contain null characters. - Added example of empty RDS string. - Added Reserved and Proprietary string types. - Added Unicode to the UTF8 lines and UTF16 to the Unicode lines. - Removed the ASCII code type.
2V2_018	2.3.2.7.12	- Classified Stream type added.
2V2_019	2.3.4	- Removed the paragraph that provided information about Protocol Catalogs in OASIS tools.
2V2_020	2.3.5	- Changed instance ID to 1. Since default instance ID was changed.
2V2_021	2.3.11.1	- Changed to a table. - Added Channel Type and Reserved bitfields. - Added that Unicode is not ASCII compatible - Added Table 2-9 and descriptions about the different modes that can be set through Channel Type. - Added Function Class Container (0x1B) - Changed 0x1A to BitSet
2V2_022	2.3.11.1.2	- Changed mils to miles in Table 2-10.
2V2_023	2.3.11.1.7	- Added Function Class Container.
2V2_024	2.3.11.2.3 2.3.11.2.4.2	- Changed <> to = in front of "0x01not allowed, no access to Tag"
2V2_025	2.3.11.2.4.3	- Clarified that parameters are not used in mode Top and Bottom. - Clarified what happens when an invalid position of the ArrayWindow is reached.
2V2_026	2.3.11.2.4.4	- Added Re-synchronization of ArrayWindows.
2V2_027	2.3.12	- Removed the requirement of three entries.
2V2_028	3.2.2.3	- Added text and Figure 3-7 to better explain how devices behave when unlocks occur.
2V2_029	3.2.2.4	- Changed timer from t_{Lock} to t_{Diag_Lock}
2V2_030	3.2.3.1	- Added chapter about Configuration Status Events
2V2_031	3.2.3.2	- Added information about t_{Bypass} which is set to 200ms. - Changed Figure 3-14 to include a new timer.
2V2_032	3.2.3.3	- Removed the requirement to store the Decentral Registry in buffered RAM.
2V2_033	3.2.5.1	- Added information about electrical wakeups.
2V2_034	3.2.6.3	- Added a sentence to explain that the behavior applies to all sinks.
2V2_035	3.2.6.4	- Added that the status message may not be sent before the NetworkMaster has asked the device.
2V2_036	3.2.6.6	- Removed the Power Save Mode and altered the text to fit the new structure. - Figure 3-21 was redrawn.
2V2_037	3.2.7	- Included a chapter about Over-Temperature Management.
2V2_038	3.3.1	- Changed the text about Node Position Address.
2V2_039	3.3.5.3	- Changed a "nein" to "no" in Figure 3-23.
2V2_040	3.3.6	- Added a "yes" to Figure 3-24.
2V2_041	3.3.7.2	- Added a maximum length to segmented transfers.
2V2_042	3.4.1.1.2.1	- Changed the second parameter to RequiredChannels. - Added that allocation must not be done partially.
2V2_043	3.4.1.1.3	- Added "Handling of Handling of Double (De)/Allocate/(Dis)Connect Commands"
2V2_044	3.5.2.1	- Added TelID "B" for MAMAC48
2V2_045	3.5.3	- Removed chapter about MAMAC and included a short overview and a reference to [9].
2V2_046	3.7.1.2	- Point 5 was updated to RequiredChannels. Point 7 was removed.

Change Ref.	Section	Changes
2V2_047	3.8	- t_{Config} changed to 2000ms. - t_{Bypass} added. - $t_{WaitAfterNCE}$ changed to 200ms, and it is now a minimum - Added the Sentence “This time also applies to a shutdown after a slave-wakeup.” to $t_{ShutDown}$. - $t_{Restart}$ changed to 300 ms or MPR*15 ms. Added “The 300ms minimum applies to networks containing up to 20 devices. For larger networks the time can be calculated as follows: $t_{Restart}$ > MPR * 15ms”. - $t_{DelayCfgRequest}$ added to complement the Figure 3-14.
2V2_048	5	- This chapter was removed.

Changes MOST Specification 2V0-01 to MOST Specification 2V1-00

Change Ref.	Section	Changes
2V1_001	1	- Added paragraph introducing object oriented approach
2V1_002	2.3.2.2	- FBlockIDs “System Specific” and “Supplier Specific” added (WG-DA 2000-09-12)
2V1_003	2.3.2.4	- FktIDs “System Specific” and “Supplier Specific” added (WG-DA 2000-09-12) - Handling of proprietary Functions/ Function Blocks by controller added (WG-DA 2000-02-09)
2V1_004	2.3.2.2	- Speech output Device added (WG-DA 2000-09-12) - Speech Database Device added (WG-DA 2000-09-12) - Corrected FBlockIDs DAB Tuner (0x43) and TMCTuner (0x41) - FBlock Satellite Radio (0x44) added - FBlock HeadphoneAmplifier (0x23) added - FBlock AuxiliaryInput added (0x24) - FBlock MicrophoneInput added (0x26) - FBlock (0x51) "Telephone mobile" replaced by "Phonebook" - FBlock Router added (0x8) (WG-DA 2001-01-17)
2V1_005	2.3.2.5.1	- Specification of “Error Secondary node” revised - Specification of “Error Device Malfunction” added (WG-DA 2000-05-04) - Specification of “Segmentation Error” added (WG-DA 2000-09-12) - Hint to avoiding “infinite loops” added - "No error replies allowed in case of reception of broadcasted messages" added - Specification of “ Error Method Aborted” added (WG-DA 2000-11-22) - Added remark that methods in general should be aborted only by that application, which has started the method. - Code 0x05 and 0x06: Returning of the value of first incorrect parameter is optional (WG-DA 2001-01-17).
2V1_006	2.3.2.5	- Renamed StartAck -> StartResultAck (0x6) and adapted every occurrence in specification document. - Added AbortAck (0x7) - Added New StartAck (0x8)
2V1_007	2.3.2.5.4	- Added New StartAck (0x8)
2V1_008	2.3.2.5.11	- Added AbortAck (0x7)
2V1_009	2.3.2.6	- Maximum value for LENGTH changed to 65535
2V1_010	2.3.2.7	- Encoding of signed values added - Codes for ISO 8859/15 8 bit and UTF8 added - Maximum value for LENGTH changed to 65535 - Examples enhanced - Data type Boolean revised - Data type BitField added - Description of String enhanced (Null Strings)
2V1_011	2.3.2.5.3	- Flow chart “Flow for handling communication of methods (controller’s side)”. Error handling for “Timeout = YES” added - Changing of timeout (100ms) for "PROCESSING"
2V1_012	2.3.11.1.2	- Specification of NSteps extended - Units for Speed (m/s), Angle and Pixel added
2V1_013	2.3.11.1.4	- Interpretation of Increment and Decrement added
2V1_014	2.2.6	- Handling of dynamic changes of Function Interfaces through Notification added
2V1_015	General	- MMI replaced by HMI (Human Machine Interface)
2V1_016	3.9	- Description of Secondary Node added
2V1_017	3.2.6.8	- Section completely removed, due to an overlapping with the MOST Function Catalog

Change Ref.	Section	Changes
2V1_018	3.2	- Generally revised
	3.2.2	- Figure 3-3 “Diagnosis Normal Shutdown” changed to “Diagnosis Ready”
	3.2.2.1	- Table 3-6 changed
	3.2.3.2	- “Network Slave” removed, “Requesting System Configuration – Network Master” added
	3.2.3.3	- “Network Master” removed, “Requesting System Configuration – Network Slave” added
2V1_019	3.2.4	- Dynamic Behavior of Secondary Nodes added
2V1_020	3.2.6.4	- "Failure Of A Function Block" added
2V1_021	3.8	- Timeout $t_{Runtime}$ added - Timeout $t_{CfgStatus}$ changed - Timeout t_{Answer} changed - Timeout t_{Diag_Master} changed - Timeout t_{Diag_Slave} changed
2V1_022	2.3.12	- Error handling in case of property failure added - Notification of Function Interface (FI) added (WG-DA 2001-01-17) - Error handling added, in case of values in property not yet available during subscription (WG-DA 2001-01-17)
2V1_023	2.3.2.2	- Note about FBlockID 0xFF added
2V1_024	2.3.11.2.4.2	- Added parameters CurrentSize and AbsolutPosition to description of ArrayWindows - Added PositionTag, and descriptions for PositionTag and WindowSize (WG-DA 2001-01-17)
2V1_025	2.3.2.5.10	- Added remark that methods in general should be aborted only by that application, which has started the method.
2V1_026	2.3.2.5.11	- Function Class BoolField added - Function Class BitField added - Description of parameter "OPType" enhanced
2V1_027	2.3.11.1	- Start of Ring Break Diagnosis revised
2V1_028	3.2.5.1	- Note about wakeup methods added
2V1_029	4.6, 4.7	- Voltage levels and Implementation of Power Supply Area are no longer normative.
2V1_030	General	- Eithernet replaced by Ethernet
2V1_031	3.2.2.2	- Behavior of a waking Slave device (Figure 3-6)
2V1_032	3.5.3	- “MOST Asynchronous Medium Access Control (MAMAC)” added
2V1_033	3.4.3.3	- Equation for delay compensation revised ($T_{Source} < T_{Node}$)
2V1_034	4.2.4	- Hint Added. Description of pig tail is only one of the possible implementations.
2V1_035	3.4.1.1.2.1	- Method SourceActivity added
2V1_036	3.2.6	- General handling of errors. Synch. connections are removed in case of Fatal Errors.

Changes MOST Specification 2V0-00 to MOST Specification 2V0-01

Change Ref.	Section	Changes
2V01_001	General	Document no longer specified as “Confidential”; Legal Notice inserted.

Changes MOST Specification 1V0 to MOST Specification 2V0

Change Ref.	Section	Changes
2V0_001	3.3.1	Equation modified; Startup address 0xFFFF
2V0_002	2.1.2/ 2.2.5	Section 2.2.5 moved to 2.2.2
2V0_003	2.2.1	NetBlock “functions related to the entire device.”
2V0_004	2.3.2.2	Table 2-5: Proprietary FBlockIDs 0xF0..0xFE
2V0_005	2.3.2.3	Completely revised
2V0_006	2.3.2.4	Minor modification
2V0_007	2.3.2.5	Completely revised
2V0_008	2.3.2.6	Completely revised
2V0_009	2.3.2.7	Boolfield introduced; Definition of STRING expanded, Examples for Exponent, Step and Unit
2V0_010	2.3.5	Minor modification
2V0_011	2.3.6	Distinguishing Properties and Methods; Communication with routing revised
2V0_012	2.3.10	Transmitting function interfaces. Introduced.
2V0_013	2.3.11	Function Classes (completely revised)
2V0_014	2.3.12	Notification for array properties; Notification re-build at system start
2V0_015	3.2.2.2	Error_t_slave replaced by Error_NSInit_Timeout
2V0_016	3.2.2.3	Completely revised
2V0_017	3.2.2.4	Completely revised
2V0_018	3.2.3.2	Completely revised
2V0_019	3.2.3.2	Completely revised
2V0_020	3.2.5.1	Completely revised
2V0_021	3.2.6	General rules added
2V0_022	3.2.6.1	Completely revised
2V0_023	3.2.6	Completely revised
2V0_024	3.3.5.3	- Table 3-14; - sample for receiving logical node address; - section below Table 3-15
2V0_025	3.3.7.2	TelIDs for MOST High Protocol removed
2V0_026	3.3.8.1	Figure 3-25; Set STX bit added
2V0_027	3.4.1.1.1	Replaced “.0.” by “.Pos.”
2V0_028	3.4.1.1.2	Completely revised
2V0_029	3.4.3.3	Equations
2V0_030	3.5.2.1	TelID and TelLen changed; One ID reserved for Ethernet frames
2V0_031	3.7.1.1	Revised (OPTypes)
2V0_032	3.7.1.2	Revised (OPTypes)
2V0_033	3.7.1.3	Revised (OPTypes)
2V0_034	3.1.4.2	Table 3-1
2V0_035	3.1.4.2.2	Revised
2V0_036	3.1.4.3.1	Handling of Isochronous data removed

Change Ref.	Section	Changes
2V0_037	3.1.4.3.6	Table 3-2; Table 3-3 added, Handling of Isochronous data removed
2V0_038	3.1.4.4.2	Completely revised
2V0_039	4.1	Figure 4-1
2V0_040	4.2.1	Completely revised
2V0_041	4.2.2	Revised
2V0_042	4.2.4	Completely revised
2V0_043	4.3	Revised
2V0_044	4.5	Completely revised
2V0_045	4.6	Completely revised
2V0_046	4.7	Completely revised
2V0_047	—	General changes in Structure: - Chapter 2.1 removed, contents included within 2.2.9 - Detailed descriptions of Control Channel (2.2) moved to 3.3 - Introduction of CMS/ AMS moved to 3.3.7 - Chapters 2.6 up to 2.12 moved to 3.2 up to 3.8 - Chapter 2.5 and 2.13 moved to Chapter 5

Notes:

Notes:

Notes: