MOST

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

**Multimedia and Control
Networking Technology**

**MOST Specification
Rev 2.4
05/2005**

COOPERATION

Legal Notice

SUPPORT AND FURTHER INFORMATION

For more information on the MOST technology, please contact:

MOST Cooperation
Administration
P. O. Box 4327
D-76028 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

MOST®
COOPERATION

Contents

Document History

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	Removed sentence about asynchronous channel administration.
2V4_004	2.2.1	Changed description for CD player.
2V4_005	2.2.8	System Service changed to Network Service.
2V4_006	2.3.2.5	Complemented table.
2V4_007	2.3.2.5.1	New wording for Segmentation error with Error Info 0x04.
2V4_008	2.3.2.5.1	Clarified ErrorCode 0x01.
2V4_009	2.3.2.5.1	Reserved error code for supplier specific error codes.
2V4_010	2.3.2.5.1	Improved language
2V4_011	2.3.2.5.1	For clarification, ErrorCode 0x02 is also explained
2V4_012	2.3.2.5.1	Error codes removed 0x08 and 0x09 removed from paragraph “Application error – Parameter error”.
2V4_013	2.3.2.5.1	Added examples to ‘Syntax error’ and ‘Error secondary node’.
2V4_014	2.3.2.5.1	Updated figure 2-15.
2V4_015	2.3.2.5.5	Added timer $t_{WaitForProperty}$.
2V4_016	2.3.2.5.7	Added timer $t_{WaitForProperty}$.
2V4_017	2.3.2.5.10 2.3.2.5.11	Changed description for Abort and AbortAck.
2V4_018	2.3.2.7 2.3.2.7.13	Added data type short stream.
2V4_019	2.3.2.7.2	Representation of Boolean Data Types
2V4_020	2.3.2.7.12	Clarification that there is no coding Byte before the string and the string is always coded in ASCII.
2V4_021	2.3.5 2.3.6	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.
2V4_023	2.3.11.2.1 2.3.11.2.2	Updated tables that describe IntDesc.
2V4_024	2.3.11.2.3	Deleted EI4 in example.
2V4_025	2.3.11.2.4	Completed tables with lost data.
2V4_026	2.3.11.2.4.2	Changed wording for ArrayWindow.
2V4_027	2.3.11.3.2	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	3.1.1	Section name Electrical Bypass changed to only Bypass.
2V4_031	3.1.1	Changed description for electrical bypass.
2V4_032	3.1.5.1	Reserved device address 0x0FF0 as optional for debug purpose.
2V4_033	3.1.4.3.4 3.2.2.2 3.2.2.3 3.2.2.4 3.6.1.1 6	Removal of MOST transceiver register references.

Change Ref.	Section	Changes
2V4_034	3.2.2.2 3.2.5.2.1	Replaced t_{Master} and t_{Slave} with t_{Config}.
2V4_035	3.2.2.4	Replaced t_off by t_restart in Figure 3-13.
2V4_036	3.2.4	Changed description.
2V4_037	3.2.4.2 3.9	Added timer $t_{SlaveShutdown}$.
2V4_038	3.2.4.3.1	Increased description for “Request Stage”.
2V4_039	3.2.5.4	New definition of NCE.
2V4_040	3.3.2.2	Wording changed in bullet number 4.
2V4_041	3.3.2.2	The Connection Manager must not de-allocate channels.
2V4_042	3.3.3.1.2 3.3.3.3.4 3.9	Changed timer $t_{WaitAfterNetOn}$ to $t_{WaitBeforeScan}$ and extended timer to cover Configuration.Status(NotOk).
2V4_043	3.3.3.4.7	Changed description.
2V4_044	3.3.3.6.3	Changed the headline.
2V4_045	3.3.3.6.3	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	3.3.4.3.13	Chapter updated.
2V4_047	3.4.1	Group Address part extended.
2V4_048	3.4.1	Four changed to five.
2V4_049	3.4.2	Reduced chapter about control message priority.
2V4_050	3.5.2.1	Improved language.
2V4_051	3.5.2.2	Channel lists must always be in ascending order.
2V4_052	3.5.2.2.1	Increased requirements for Connection Manager.
2V4_053	3.5.2.2.3	Sink changed to source.
2V4_054	3.7	Deleted part that describes Boundary.
2V4_055	3.8.1.1	InstID changed to 1.
2V4_056	3.8.1.1	Extended parameter lists.
2V4_057	3.8.1.2	ResultAck changed to Result.
2V4_058	3.9	Added new timer $t_{WaitForProperty}$.
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

Change Ref.	Section	Changes
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 function block 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.
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, Aplication may PowerOff in “Device Standby”. Voltage levels are no longer exactApplication 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.

Change Ref.	Section	Changes
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.
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.

Change Ref.	Section	Changes
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.
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
2V1_018	3.2	- Generally revised

Change Ref.	Section	Changes
	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
2V0_037	3.1.4.3.6	Table 3-2; Table 3-3 added, Handling of Isochronous data removed

Change Ref.	Section	Changes
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

1 Introduction

1.1 Purpose

The purpose of this documentation is to be part of the MOST (Media Oriented System Transport) specification. This document is the main specification, which all other specifications relate to.

1.2 Scope

This document contains specification of the application layer, the network layer and the MOST Hardware.

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 changes as new documents are published.

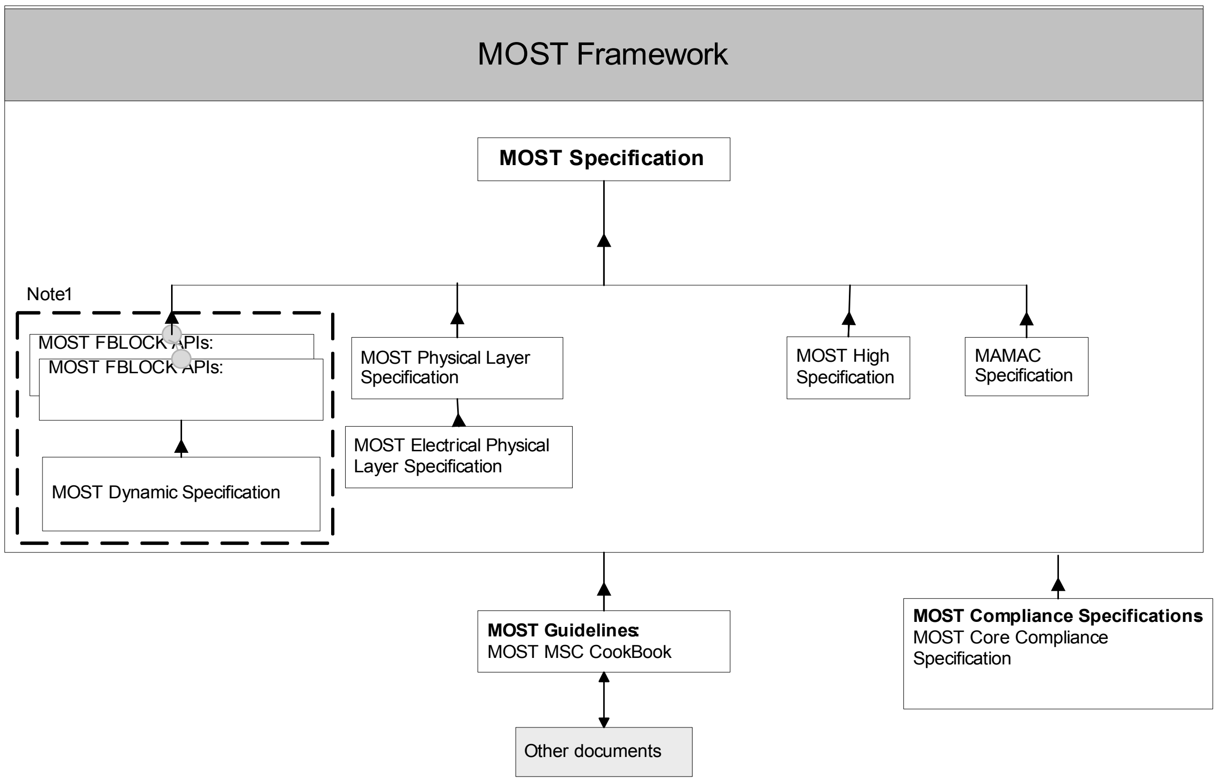

MOST Framework
MOST Specification
Note1
MOST FBLOCK APIs:
MOST FBLOCK APIs:
MOST Dynamic Specification
MOST Physical Layer Specification
MOST Electrical Physical Layer Specification
MOST High Specification
MAMAC Specification
MOST Guidelines: MOST MSC CookBook
MOST Compliance Specifications MOST Core Compliance Specification
Other documents

Figure 1-1: MOST Document Structure

Note1: MOST FunctionBlocks API's will also include the dynamic behavior. A restructuring taking this into account is planned.

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

1.4 References

All documents within this MOST document have references to are listed here with the actual revision this document is referring to.

Document	Revision
-	-
-	-

Comment: The MOST Specification does not refer to other documents.

1.5 Overview

This specification consists of three sections namely the application section, the network section and the hardware section. There are different possible physical layers described in 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

2.1 Overview of Data Channels

2.1.1 Control Channel

On the control channel, data packets are transported to certain addresses, as they are on the asynchronous (packet) channel. Both channels are secured by CRC.

The control channel also has an ACK/NAK mechanism with automatic retry. It is generally specified for event-oriented transmissions at low bandwidth and short packet length. It is usable for connections with a bandwidth of approximately 10KBps, even for short periods of time.

In contrast to that, the asynchronous area is specified for transmissions requiring high bandwidth in a burst-like manner.

2.1.2 Synchronous Channel

Continuous data streams that demand high bandwidth are transported over the synchronous channels. The connections are administered dynamically via the control channel. No bandwidth is reserved for special applications. Although synchronous connections can be built directly by source and sink nodes, it is recommended that available bandwidth be administered in a central manner, particularly in larger networks.

Administration of the synchronous resources is handled by the Connection Manager. Since the Connection Manager must check to see if the connection already exists before it can be built, all requests for establishing connections must be directed to the Connection Manager. The Connection Manager may be controlled through FBlock Connection Master, but it may also be controlled in some other way.

2.1.3 Asynchronous Channel

The asynchronous channel is mainly used for transmitting data with large block size and high demand for bandwidth in a burst-like manner (graphics, some picture formats and navigation maps).

2.1.4 Managing Synch./Async. Bandwidth

On the MOST Network there are 60 Bytes available for synchronous and asynchronous data transfer. It is possible to divide up these resources between synchronous and asynchronous channels by means of a boundary descriptor. The boundary descriptor can be modified either by direct access to the respective register in a MOST Network Interface Controller, or by the MOST Network Service.

The position 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 Timing Master. The Timing Master 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. After having changed the boundary descriptor, the synchronous connections must be re-built.

2.2 Logical Device Model

The following sections describe different kinds of devices. A MOST device is a physical unit, which can be connected to a MOST network via a MOST Network Interface Controller.

2.2.1 Function Block

On the application level, a MOST device contains multiple components, which are called function blocks, e.g., tuner, amplifier, or CD player. It is possible that there are multiple function blocks 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 function blocks, which represent applications, each MOST device has a special function block called the NetBlock. The NetBlock provides functions related to the entire device. Between the function blocks and the MOST Network Interface Controller, Network Service forms an intermediate layer providing routines to simplify the handling of the MOST Network Interface Controller.

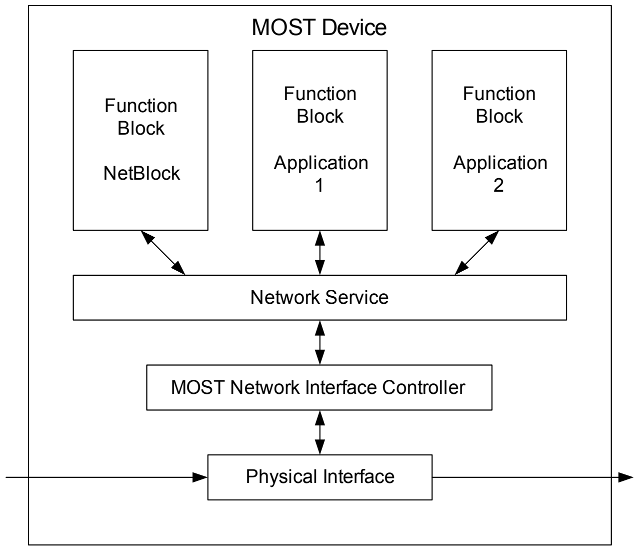

MOST Device
Function Block NetBlock
Function Block Application 1
Function Block Application 2
Network Service
MOST Network Interface Controller
Physical Interface

Figure 2-1: Model of a MOST device

Each function block contains a number of single functions. For example, a CD player possesses functions such as Play, Stop, Eject, and Time Played. To make a function accessible from outside, the function block provides a function interface (FI), which represents the interface between the function in a function block and its usage in another function block.

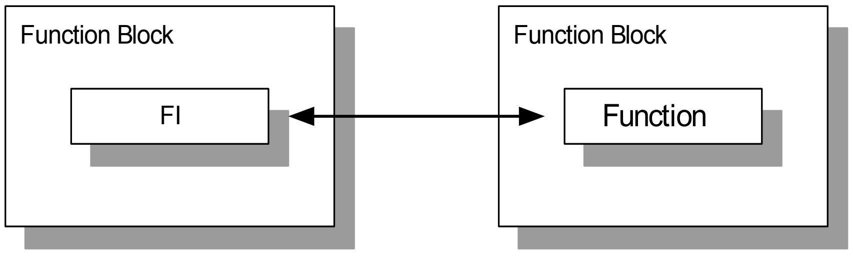


Figure 2-2: Communication with a function via its function interface (FI)

2.2.1.1 Slave, Controller, HMI

There are three types of different function blocks. Function blocks that are always controlled are called slaves. Function blocks that have an interface to the user are called Human Machine Interfaces (HMIs). Function blocks using functions in other function blocks are called controllers. Controllers themselves may also be controlled.

A clear separation between HMI, controller and slave cannot always be made in devices that have many function blocks. Such devices can often be classified with respect to their primary function.

2.2.1.2 First Introduction to MOST Functions

This section gives a brief introduction to the structure of MOST functions, as this knowledge is necessary to understand the following examples. Chapter 2.3 on page 38 explains the structure of MOST functions in more detail.

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

FBlockID . FktID

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

FBlockID . FktID . OPType (Data)

2.2.2 Functions

A function is a defined property of a function block that can communicate with the external world, through the borders of its function block. Functions can be subdivided into two classes:

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

In addition to that, there are also events. Events result from properties, if the properties are requested to report changes (Notification).

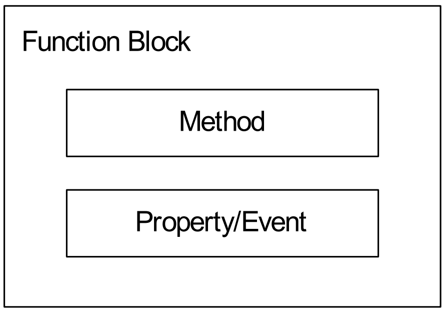


Figure 2-3: Structure of a function block consisting of functions classifiable as methods, properties and events

2.2.3 Methods

Methods can be used to control function blocks. They are transmitted in the same way as properties. In general, a method is triggered only once, for example, starting the auto-scanning of a tuner. So method “auto-scan” is started without parameters. Of course, it is possible to use parameters, e.g., to specify the direction of auto-scan. Then only one method is needed for tune up and tune down. Especially in the case of tuners that possess automatic frequency optimization (RDS) it may be useful to specify the starting frequency for the scanning process as an additional parameter, since the currently displayed frequency may no longer meet the frequency the receiver is tuned to. So a method’s call may contain one or more parameters.

After the reception of a method called by a function block, the respective process must be started. If this is not possible, the function block has to return the respective error message to the sender of the method call. This may happen if the addressed function block has no method of that kind, if a wrong parameter was found, or if the current status of the function block prevents the execution of the method.

After finishing the process, the controlled function block should report execution to the controlling function block (in a control device). 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 controlling function block 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 (*Start/StartResult*)	Error with cause for error (*Error*) Execution report with results (Result) Intermediate result (Processing)

The respective MOST functions needed for this messaging are described in depth in chapter 2.3 on page 38.

2.2.4 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 a function block, a property is normally represented by a variable that represents something such as a limit, or a status.

2.2.4.1 Setting a Property

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

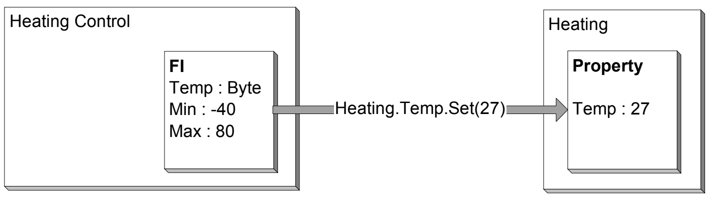


Figure 2-4: Setting a property (temperature setting of a heating)

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

2.2.4.2 Reading a Property

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

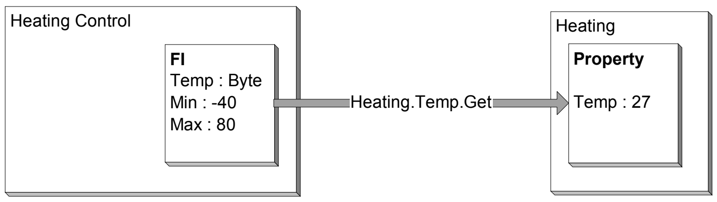


Figure 2-5: Reading a property (temperature setting of a heating)

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

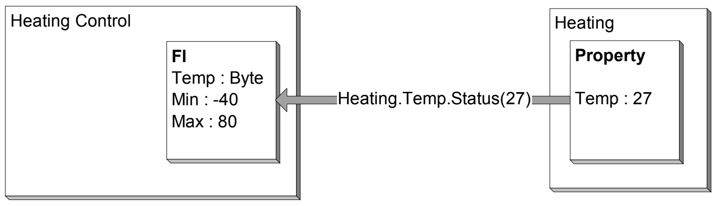


Figure 2-6: Status report of property temperature setting

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

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

The MOST functions needed for this messaging are described in depth in chapter 2.3 on page 38.

2.2.5 Events

Properties of a function block may change without an external influence, e.g., 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 between function blocks, it would be useful if function blocks could send status reports about changes in properties without explicit requests. These are events that occur in a controlled function block, which initiate the sending of a report (notification).

Events can be used to notify reaching of limits, or the change of measured values in function blocks (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). Events are sent only to those function blocks that have requested it by an entry in the notification matrix (refer to chapter 2.3.12 on page 107). The respective data should be transmitted in the same message with events.

2.2.6 Function Interfaces

A function interface (FI) represents the interface between a function in a function block, and its use in another function block.

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, e.g., 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 programming language C.

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 a HMI. In this way, even functions that did not exist during the development of a HMI can be made available.

Example:

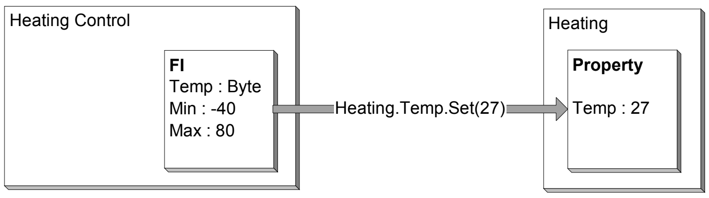


Figure 2-7: Example for a function interface (FI)

In this example the FI contains information about the data type of the function and about minimum and maximum value. In real implementations, a FI contains much more information.

During operation it is possible that a FI changes dynamically. In that case, all the function blocks that have subscribed for notification, will get the new interface description through the notification mechanism. For more information about notification, please refer to section 2.3.12 on page 107.

2.2.7 Definition Example

This section contains the example of a formal definition of a MOST device, MyTuner, its function blocks and their methods and properties.

```
MyTuner = Device                            Device
            Tuner : TTuner;                   {
            NetBlock : TNetBlock;                TTuner Tuner;
          end;                                   TNetBlock NetBlock;
                                              } MyTuner
```

(Pascal Syntax) (C Syntax)

The definition specifies that MyTuner contains a function block Tuner of type TTuner, and a function block NetBlock of type TNetBlock.

TTuner is a function block and can be defined in the following way:

```
TTuner = Object                             Object
           pStation    : TStation;            {
           eTraffic    : TTraffic;              TStation pStation;
           pSensitivity: TSensitivity;          TTraffic eTraffic;
           mSearch     : TSearch;               TSensitivity pSensitivity;
         end;                                   TSearch mSearch;
                                              } TTuner
```

Here it is defined that function block TTuner contains the functions pStation (currently tuned station), pSensitivity, and mSearch (auto scan). In addition to that, the event eTraffic can be generated.

The type of function can be indicated by its name, by adding a special character to the beginning of the name (p = property, m = method, e = event).

Now property pStation will be defined as follows:

```
TStation = Property                         Property
             Frequency : Long;                {
             TP        : Bool;                  long Frequency;
             Quality   : Byte;                  Bool TP;
           end;                                 Byte Quality;
                                              } TStation;
```

This describes pStation as a property with the parameters Frequency, TP, and Quality.

Now method mSearch will be defined:

```
TSearch = Method                            Method
             Up : Bool;                       {
             Start : Long;                      Bool Up;
           end;                                 Long Start;
                                            } Tsearch;
```

Method mSearch can be started with the parameters Up for direction and Start for the start frequency.

And last, the definition of event eTraffic:

```
TTraffic = Event                              Event
             TA : Bool;                         {
           end;                                   Bool TA;
                                              } TTraffic;
```

This definition specifies that event eTraffic has a Boolean parameter TA (traffic announcement).

The FI of property pStation could be defined as follows:

```
iStation = Interface
             iFrequency,
             iTP
             iQuality
           end
```

Here is the example for the interface description of parameter Frequency.

```
iFrequency = Interface
               Type : Tlong;
               Min  : 87500
               Max  : 108000
               Unit : TKHz
             end
```

The interface description of property TStation, consisting of interface definitions for the parameters Frequency, TP, and Quality, can be available as part of the device specification and can be regarded as a well-known function. It can also be requested by the control device and sent to it encoded in a suitable form.

2.2.8 MOST Network Service

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 or properties are mandatory for a MOST device. MOST devices should be able to handle control tasks in a peer-to-peer manner. To provide flexibility in control tasks, MOST devices must be able to work in an environment with multiple masters.
MOST Network Service provides a basic framework for a MOST device.

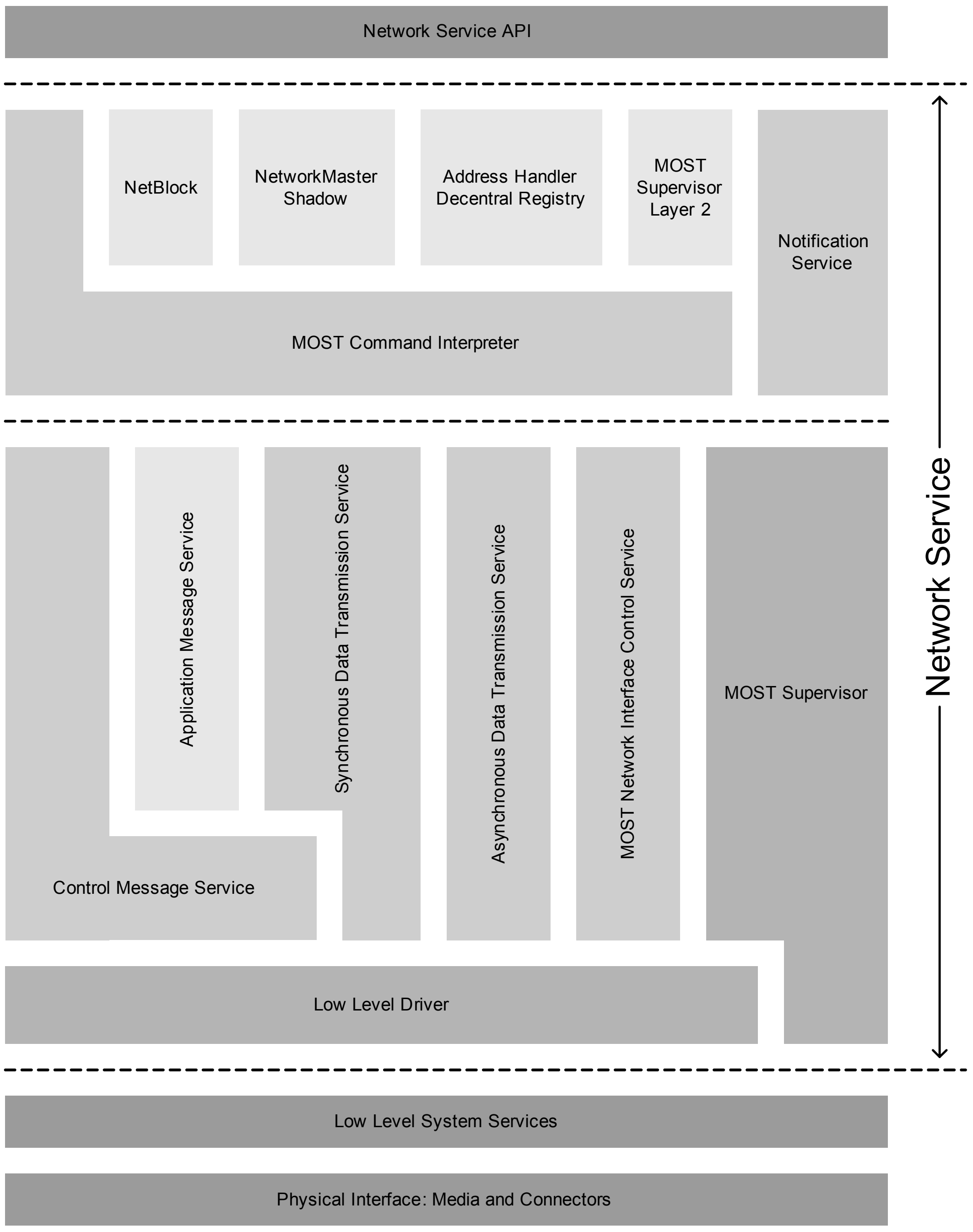

Network Service API
NetBlock
NetworkMaster Shadow
Address Handler Decentral Registry
MOST Supervisor Layer 2
Notification Service
MOST Command Interpreter
Application Message Service
Synchronous Data Transmission Service
Asynchronous Data Transmission Service
MOST Network Interface Control Service
MOST Supervisor
Network Service
Control Message Service
Low Level Driver
Low Level System Services
Physical Interface: Media and Connectors

Figure 2-8: MOST Network Service

2.2.9 Delegation, Heredity, Device Hierarchy

2.2.9.1 Delegation

The principle of delegation provides the combining of functions of several devices to higher, distributed functions. By combining tasks and by simplifying presentation in the direction to upper layers, device hierarchies are built, which allow higher-level software to structure and control even complex system contexts in a clear way. The following example illustrates delegation.

Although car audio systems today consist of many single components, an ideal audio system would look like the one shown below, from the view of a HMI:

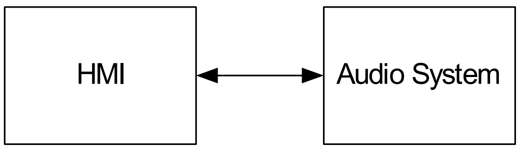


Figure 2-9: Ideal audio system

Real audio systems generally look like this:

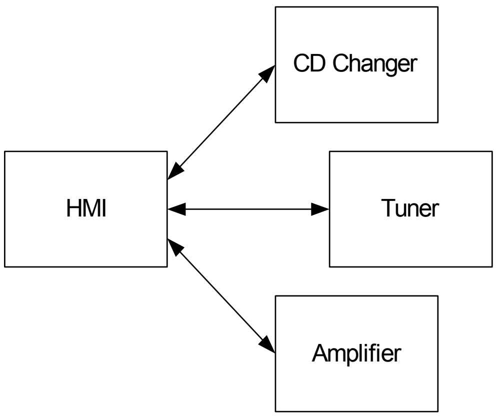


Figure 2-10: Real audio system

Coordination of the complex interaction of these components must normally be done by the HMI. This makes the design of the HMI complex and vulnerable to design changes. In addition to that, coordination of audio components requires detailed knowledge of a special range of problems. This also applies to other subsystems such as video, communication, and vehicle-based functions.

The goal of delegation is to present the audio components as one single component providing audio functions. This delegation can be related only to the direct audio area, and not to all devices having audio functionality like a navigation system or a telephone. If the audio controller were to take over complete control of these other devices, it would lead to unnecessary dependencies, making the system inflexible.

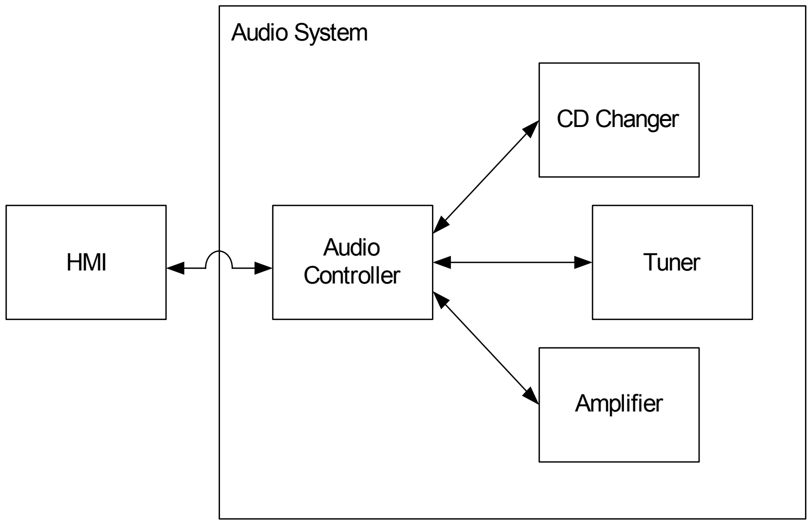


Figure 2-11: Delegation of functions of all audio components to one audio controller

By defining an audio controller, all audio functions can be provided by a single hand, even if the components are distributed. The audio controller coordinates the audio components in this case. The audio controller does not need to be a real physical device in the ring; it can be part of another device. It could even be a software module in the HMI.

The mechanism of delegation of functions is explained below by the example of a traffic announcement. The tuner device has the possibility to detect traffic announcements (TA). For a TA to be faded in during CD player plays, some steps must be taken.

Before the announcement:
Set the CD to pause, connect the amplifier to the tuner's channels (eventually send special values for volume and sound).

After announcement:
CD back to play, amplifier back to CD, restore Volume and sound.

The sequence and the timing of these operations must be done precisely, to avoid unpleasant effects for the listener. In order to keep the design of the HMI simple, these operations can be handled by a special unit, the audio controller. With this controller, a distributed higher TA function is built. The audio controller can provide a TP on/off, although the tuner has no such function. The interface to the HMI is represented by two simple functions TA and TP. This shows how control can be simplified by building hierarchies.

2.2.9.2 Heredity of Functions

The example above also shows a second mechanism - the heredity of functions. The audio controller receives function TA from the tuner and hands it through to the HMI in a modified form. The TA function of the tuner is complemented by an on/off function. TA information is only passed to the HMI in case of TP = ON.

2.2.9.3 Deriving Devices / Device Hierarchy

The principle of deriving provides a system of order, where complex devices can be derived from simpler devices. The parent node provides functions and properties to all child nodes (parent is that node from which properties are inherited, while child nodes are those which inherit). In complex devices, this heredity can also be found in a respective hierarchy of classes. All devices in lower hierarchical devices (derived devices) contain all of the functions and properties of higher device classes (mandatory functions). These functions and properties can be taken from them or can be modified. Additional properties and functions can also be defined for these objects. A kind of "constructional toy" principle is generated, where complex structures are built from simpler ones.

In the figure below, the upper layer of the device hierarchy is displayed. All "intelligent" MOST devices are derived from *MOST device* and from *NetBlock*. This means they contain their entire functionality (e.g., a MOST Network Interface Controller with the properties *NodeAddr* and *LogicalAddr*) and must support all methods and properties of NetBlock and MOST device. In addition to that, their own methods and properties are added.

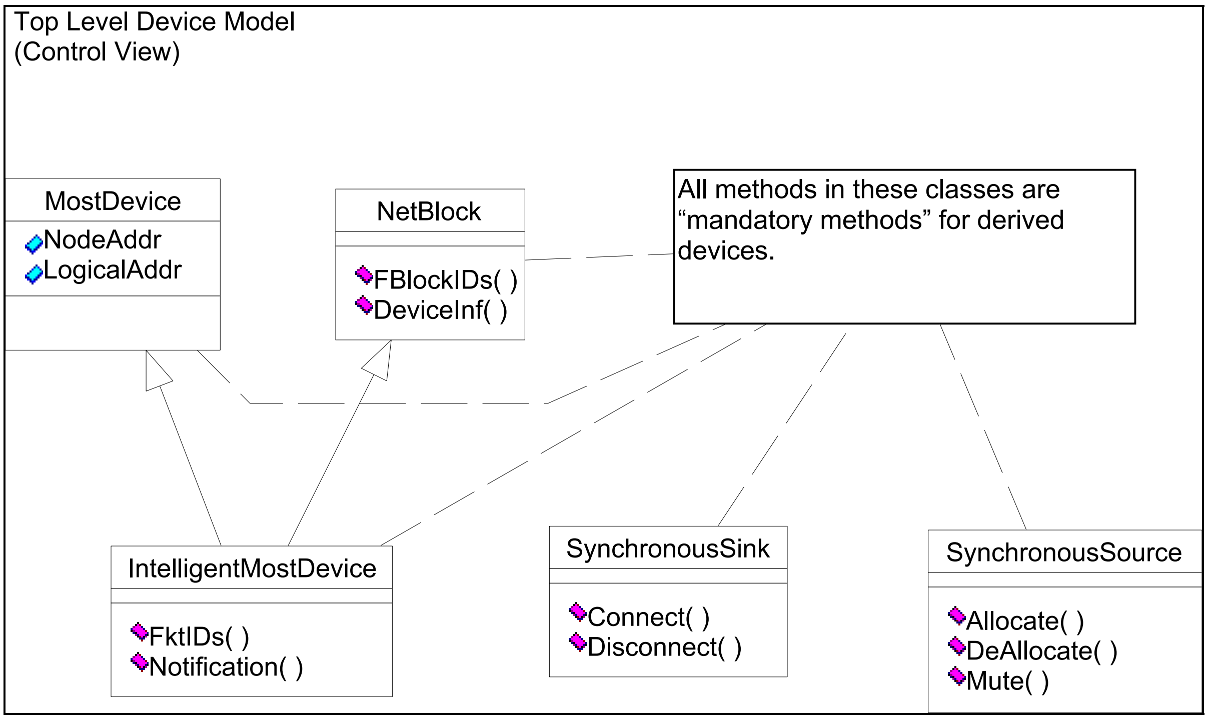


Figure 2-12: Highest layer of the device model

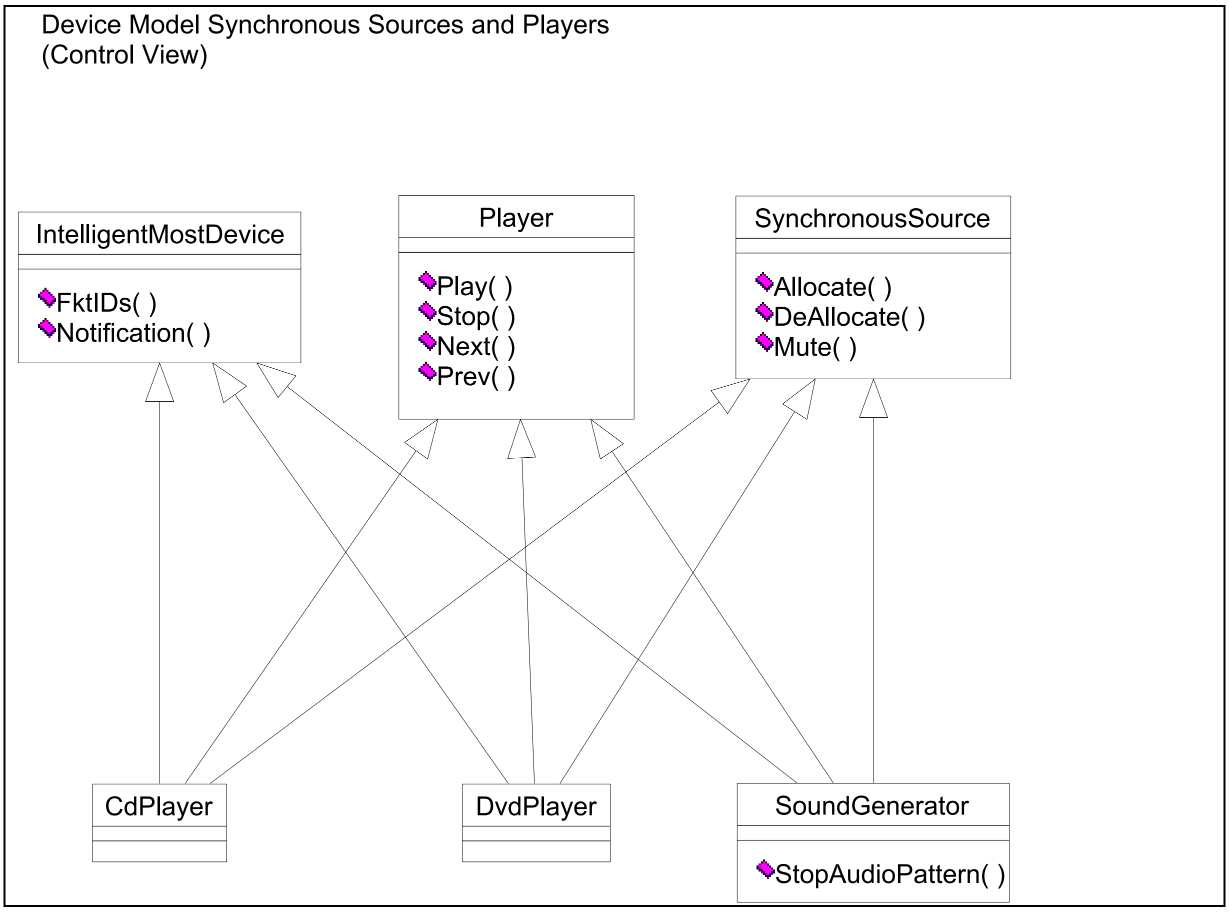


Figure 2-13: Device model for audio sources with player function

In addition to the inherited functions, the players can have their own methods and properties; for example, *StopAudioPattern* in the class *SoundGenerator*.

An application for derived devices is the designing of function or telegram catalogs for individual devices. As shown in the following table, deriving avoids re-defining all functions of the CD changer (CDMulti). They can be derived mostly from simple drives.

Device	**Group of Functions**				
Master	MOST Network Interface Controller	+ NetBlock Master			
Slaves	%	+ NetBlock Slave			
All drives	%	%	+ Basic drive		
CDSingle	%	%	%	+ CD-Functions	
CDMulti	%	%	%	%	+ Changer

Table 2-1: Application example for the principle of derived devices

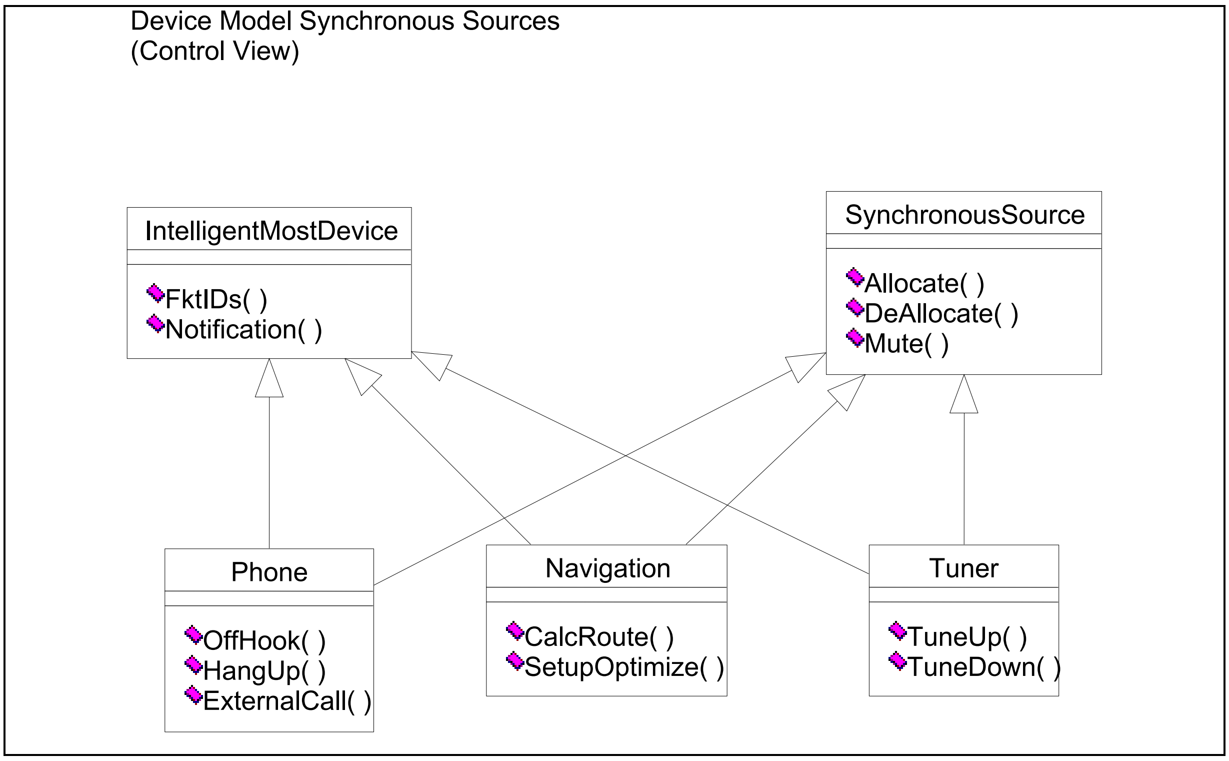


Figure 2-14: Device model for audio sources without player function

2.3 Protocols

2.3.1 Protocol Basics

As already described in section 2.2.1.2 on page 24, functions are addressed without considering the devices they belong to (on the application level). Functions are grouped together in function blocks with respect to their contents. This makes function blocks to a good reference for localizing a certain function for an external application. A function is addressed in a function block. To distinguish between the different function blocks (FBlocks) and functions (Fkt) of a device, each function and function block has a name, or an identifier (ID) respectively:

FBlockID . FktID

When accessing functions, certain operations are applied to the respective property or method. The kind of operation is specified by the OPType, followed by the parameters of the operation. This results in this structure:

FBlockID . FktID . OPType (Data)

2.3.2 Structure of MOST Protocols

The principal structure of protocols on the application layer is:

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

In addition to section 2.2.1.2 on page 24, three components were added: InstID, Length and DeviceID. The individual elements are explained below.

2.3.2.1 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 protocol, and does not need to be interpreted on the application level.

If a function receives a protocol, the DeviceID contains the logical node address of the sender (DeviceID = TxAdr = TxLog). In case of an answer, it precedes the protocol as the receiver's address (DeviceID = RxAdr = RxLog). Here a group address (DeviceID = RxAdr = GroupAddress), or the broadcast address (DeviceID = RxAdr = 0x03C8) could be used too.

If the sender does not know the receiver's address, the DeviceID is set to 0xFFFF. In that case, it is corrected by the Network Service.

2.3.2.2 FBlockID

The FBlockID is the name of a special function block. Every function block with a special FBlockID must contain certain specific functions. In addition to those mandatory functions, it can contain other functions. There are “System Specific” proprietary FBlockIDs, which can be used by any System Integrator (car maker). They are specific for a system and are coordinated between the OEMs developing devices for this system. A second kind of proprietary FBlockIDs is called “Supplier Specific”. Those FBlockIDs can be used by OEMs e.g., for development purpose. The special FBlockID 0xFF addresses all function blocks within a MOST device, except the NetBlock. Since this can be regarded as a broadcast function, no error status messages should be returned.

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

Kind	FBlockID 8 Bit	Name	Explanation
Administration	**0x0x**		
	0x00	Network Service	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	Mandatory for each system .
	0x04	PowerMaster	
	0x05	Vehicle	
	0x06	Diagnosis	
	0x08	Router	
	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	
	0x26	MicrophoneInput	
	0x28	Handsfree Processor	
Drives	**0x3x**		
	0x30	Audio Tape Recorder	
	0x31	Audio Disk Player	
	0x32	ROM Disk Player	
	0x33	Multimedia Disk Player	
	0x34	DVD Video Player	

Table 2-2: FBlockIDs (part 1)

Note: 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	
Communication	**0x5x**		
	0x50	Telephone	
	0x51	Phonebook	
	0x52	Navigation System	
	0x53	TMC Decoder	
	0x54	Bluetooth	
Video	**0x6x**		
	0x60	Display	
	0x61	Camera	
	0x62	Video Tape Recorder	
Proprietary			
	0xC0…C7	**System Specific**	
	0xC8	**Reserved**	
	0xC9...0xEF	**System Specific**	
	0xF0...0xFE	**Supplier Specific**	
	0xFC	**Secondary Node**	
	0xFE	**Reserved**	
	0xFF	**All**	

Table 2-3: FBlockIDs (part 2)

System Specific FBlockIDs (0xC0...0xC7, and 0xC9...0xEF) can be used by any System Integrator (car maker). They are specific for a system and are coordinated between the OEMs developing devices for this system. Supplier Specific FBlockIDs (0xF0...0xFE) can be used by OEMs e.g., for development purpose.

2.3.2.3 InstID

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

2.3.2.3.1 Responsibility

Each device is responsible for the uniqueness of functional addresses within the device. The Network Master is responsible for the uniqueness of functional addresses within the entire system. Refer to section 3.3.3.

2.3.2.3.2 Assigning InstID

By default, every function block has InstID 0x01. In case there are several function blocks 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 function blocks, the InstID can be chosen in any way. 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 function block. Note: Wildcard must not be used for InstID assignment.

2.3.2.3.3 InstID of NetBlock

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

2.3.2.3.4 InstID of NetworkMaster

InstID of Network Master may be zero; default value is 0x01. Requests to NetworkMaster shall be sent to InstID 0x00 (wildcard ref 2.3.2.3.6).

2.3.2.3.5 InstID of Function Block EnhancedTestability

InstIDs of function block 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).

2.3.2.3.6 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 function block in the device.
0xFF	Broadcast (within a Device). The message is dispatched to all instances of the matching function block.

Wildcards may not be used when replying to a request. In this case the correct InstID of the respective function block has to be used.

2.3.2.4 FktID

The FktID stands for a function. This means a function unit (Object) within a device, which provides operations that can be called via the network. Examples for functions are: play of a drive, speed limit in an on-board computer, etc. On network level, the FktID is encoded in 12 bits, so 4096 different methods and properties can be encoded per function block. 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 a function block.

2. **Mandatory (0x200...0x3FF)**
   Functions that are mandatory for the application of the function block, like the basic drive in all function blocks describing drives.

3. **Extensions (0x400...0x9FF)**
   Optional functions.

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

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

6. **Proprietary/ Supplier Specific (0xF00...0xFFE)**
   Functions, which can be used by OEMs e.g., for development purpose.

Some FktIDs in a function block that contains an application are predefined:

0x000	**FktIDs**	Reports the FktIDs of all functions contained in the FBlock (refer to section 2.3.9 on page 75).
0x001	**Notification**	Distribution list for events (refer to section 2.3.12 on page 107).
0x002	**NotificationCheck**	Check whether the distribution list for events is still as it should be.

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

Unique

Coordination

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

Please note:
Before using any proprietary Function or proprietary Function Block, a controller must verify the identity of the device. This can be done e.g., by reading the DeviceInfo property.

2.3.2.5 OPType

This field stands for the operation which must be applied to the property or method specified in FktID:

OPType	For Properties	For Methods
Commands:		
0	Set	Start
1	Get	Abort
2	SetGet	StartResult
3	Increment	Reserved
4	Decrement	Reserved
5	GetInterface	GetInterface
6	Locked for definitions	StartResultAck
7	Locked for definitions	AbortAck
8	Locked for definitions	StartAck
Reports:		
9	ErrorAck	ErrorAck
A	Locked for definitions	ProcessingAck
B	Reserved	Processing
C	Status	Result
D	Locked for definitions	ResultAck
E	Interface	Interface
F	Error	Error

Table 2-4: OPTypes for properties and methods

2.3.2.5.1 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.

ErrorCode Data[0] on ErrorAck Data[2]	ErrorCode Description	ErrorInfo Data[1]..Data[n] on ErrorAck Data[3]..Data[n]	ErrorInfo Description
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
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	Number of Parameter (Byte containing 1,2...). Value of first incorrect parameter only (optional). Interpretation will be stopped then.
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. Example: Function SourceHandles is asked for handle 0x03, which is not in use in the device at that time.	Return Parameter	Number of Parameter (Byte containing 1,2...). Value of first incorrect parameter only (optional). Interpretation will be stopped then.
0x08	**Reserved. Usage deprecated**	--	No Info
0x09	**Reserved. Usage deprecated**	--	No Info
0x0A	**Secondary Node**	Return Address of Primary	Address of that node which is responsible for the secondary node sending the error
0x0B	**Device Malfunction**	--	No Info
0x0C	**Segmentation Error** After this error code, the following ErrorInfo 0x01 up to 0x07 can be sent.	0x01	First segment missing
		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 sent by the same node
		0x08	Reserved, must not be used

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

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 error code, any function specific ErrorInfo can be sent. Some, with general character, are suggested here.	0x01	Buffer overflow
		0x02	List overflow
		0x03	Element overflow
		0x04	Value not available
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 error code can be used to indicate, that a method has been aborted by the Abort / AbortAck OPTypes	--	No Info
0xF0...0xFE	**Supplier specific** After this error code, any supplier specific ErrorInfo can be sent.	Optional	Supplier specific ErrorInfo.

Table 2-6: Error codes and additional information (part 2)

Please note:
The error messages described here, mainly serve the purpose of debugging. They should be handled in a controller only, if the system's performance requires it. Otherwise error processing should be omitted, and the devices should be designed as failure tolerant systems. With respect to that, the slaves also should manage with the existing error messages. Individual error messages using error code 0x20 should be avoided if possible.

Please note:
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 broadcast messages.

CAN systems often define a dedicated error value (e.g., 0xFF) for signals to indicate the failure of the sensor that provides the respective signal. If such a signal is read, function Sensor would report the error "Not available" (0x41). If the sensor fails, and the function has an implemented notification mechanism, the error is distributed to the registered controllers.

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

1) Syntax Error:
A syntax error occurs, if e.g., a function is accessed that does not exist, or if a not implemented OPType 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 with requested FBlockID, which is not available. ErrorCode 0x02 must be reported if 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)
```

2) Application Error – Parameter Error:
The specified length does not match the actual length of the data field. There have been not enough, 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. This means especially, 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 2.3.11.2 on page 89).

3) Application Error – Temporarily not Available:
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 for a method would be SMSSend of the telephone, which cannot be executed if the bus is not available. In case of being called anyhow, it would report an OPType error at error code 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 error code 0x40 “busy” would be reported. Here, the application may perform retries. This case can only occur in connection with methods.

- A property represents a memory area, which is written by Set, or read by Get. According to definition this memory area cannot be “busy”. 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. This would generate an error code 0x07 “parameter not available”.

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

5) Application Error – Specific Execution Error:
Besides the already listed errors, a MOST application may report specific errors during execution by using OPType Error as well. Here, error code 0x20 “function specific” is used. Some possible errors are predefined for that case as well.

The examination and processing of errors is done in the logical and temporary sequence as described above and in Figure 2-15 on page 49.

6) Application Error – Error Secondary Node:
Detailed information about Secondary Nodes is to be found in section 3.10 on page 200. In case a secondary node receives any control message, the requested FBlock replies with an Error „secondary node“ (ErrorCode=0x0A). The reply contains the address of the primary node (=ErrorInfo), which is responsible for that secondary node. A secondary node must report ErrorCode 0x0A with requested FBlockID, InstID and FktID, which are not available.
Example for requesting a secondary node:

```
SrcAdr -> TrgAdr:
FBlockID.InstID.FktID.OpType(...)
//if TrgAdr is secondary node
TrgAdr -> SrcAdr:
FBlockID.InstID.FktID.Error(errorcode = 0x0A, errorinfo =_)
```

7) Application Error – Device Malfunction:
This error indicates device malfunction and provides to distinguish between a generally broken device, and a temporarily being unavailable of a Device.

8) Application Error – Segmentation Error:
A MOST System provides the option of transporting messages that exceed the length limitations given by the control channel of the MOST bus (17 Bytes). This is done by dividing the message up into several segments. Each of the segments is then transported as one control channel telegram to the receiver. In order to make sure that the data can be reassembled safely on the receiver's side, each telegram carries the appropriate additional information in its protocol header (TelID, Segment counter).

Errors during reassembling the original message in the receiver can be caused e.g., 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.

The segmentation error notifies the sender about the failure of the segmented transfer. Therefore, the sender's application may react in appropriate way, e.g., by retrying to send the same message again. The reaction depends on the respective problem that caused the error.

Segmentation Error shall be sent back to a controller that failed to send a segmented message to a Slave. Error messages can only be directed from the Slave to the controller, to avoid infinite loops of error messages. So a failure in sending a segmented message from the Slave to the controller will not be notified to the Slave. In this case, it is the responsibility of the controller application to take appropriate measures, as soon as it is notified about the error by the callback function mentioned above.

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.

9) Application Error – Method Aborted:
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.

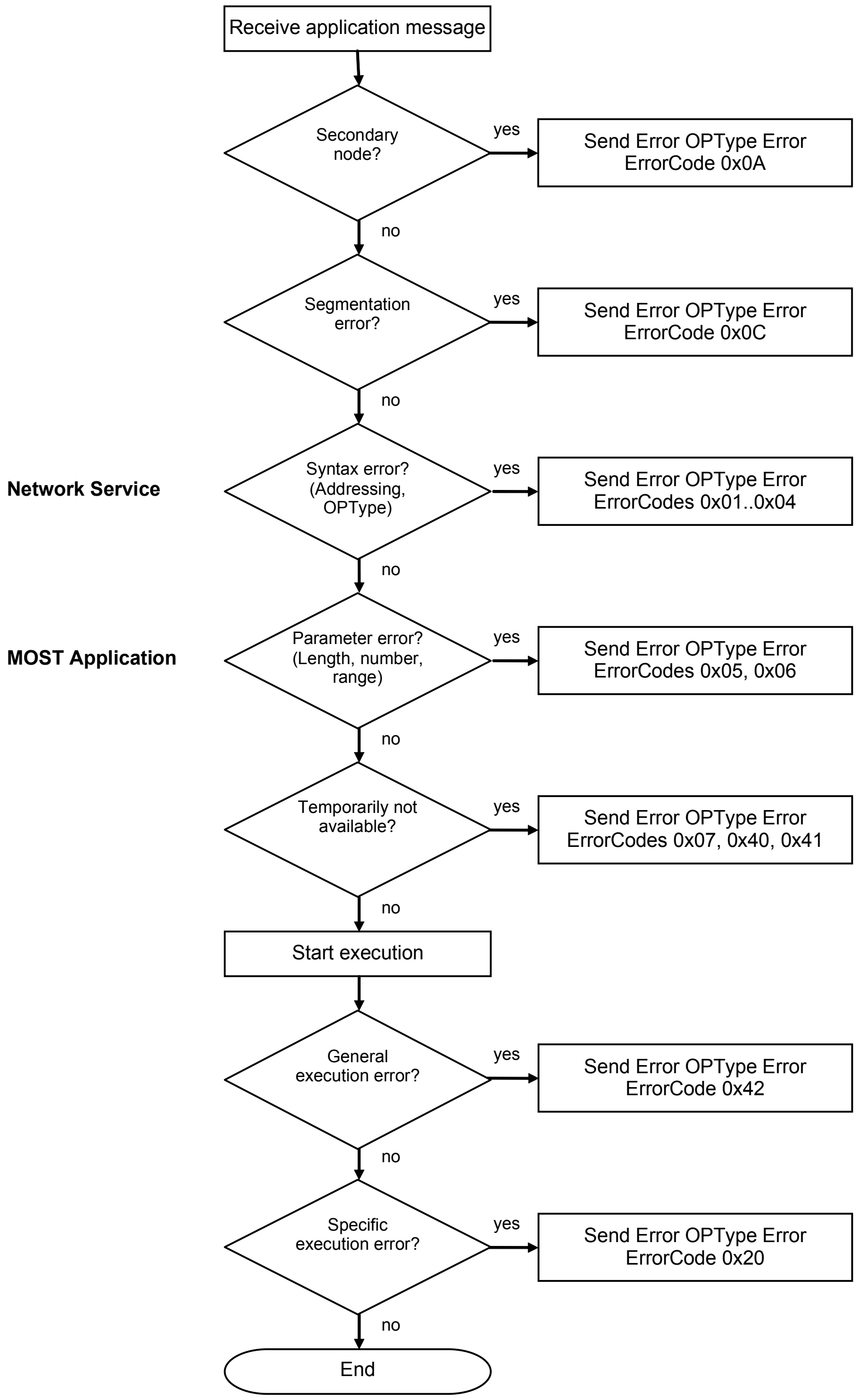

Receive application message
Secondary node?
yes
Send Error OPType Error ErrorCode 0x0A
no
Segmentation error?
yes
Send Error OPType Error ErrorCode 0x0C
no
Network Service
Syntax error? (Addressing, OPType)
yes
Send Error OPType Error ErrorCodes 0x01..0x04
no
MOST Application
Parameter error? (Length, number, range)
yes
Send Error OPType Error ErrorCodes 0x05, 0x06
no
Temporarily not available?
yes
Send Error OPType Error ErrorCodes 0x07, 0x40, 0x41
no
Start execution
General execution error?
yes
Send Error OPType Error ErrorCode 0x42
no
Specific execution error?
yes
Send Error OPType Error ErrorCode 0x20
no
End

Figure 2-15: Processing of messages including error check on different layers

Of course there exist errors on application level that do not appear in the MOST syntax (i.e. reported by OPType Error). An example would be the processing of errors within a data transfer in TCP/IP. From the point of view of the MOST system, such a data transport is only the transport of data packets to a receiving function. The contents of the packets and the fact whether that data contains errors is interpreted on application level only. Higher levels of error management and individual error messages are to be specified individually.

2.3.2.5.2 Start, Error

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

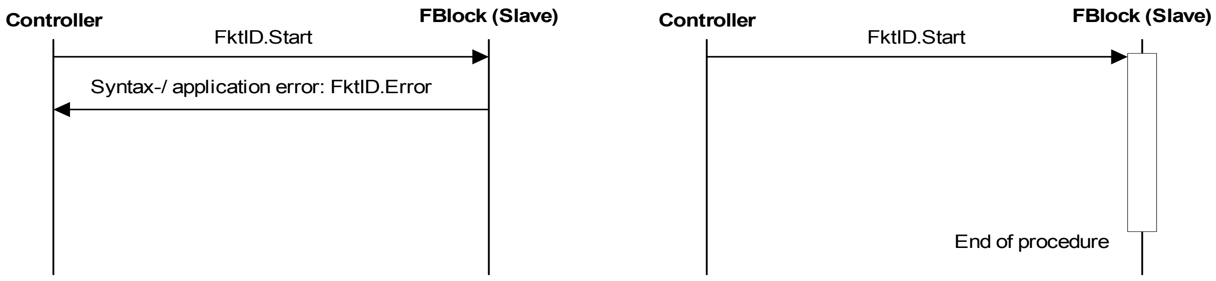


Figure 2-16: Sequences when using Start with and without error

Please Note:
A method started by "Start" must be called only one time (no multiple instances are 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 an error "Busy". The already running method is not affected by this new incoming request. For running several instances of the same method, StartAck and ResultAck must be used.

2.3.2.5.3 StartResult, Result, Processing, Error

In opposite of triggering a method by using Start, the controller requires feedback when it uses StartResult. It then expects reports about the currently running procedure (with Processing), as well as about the Result (Result or error). If a method does not return a result by parameters, it returns Result() as a signal of a successful processing.

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). In case that 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.

System Integrators may change the default timeout value ($t_{ProcessingDefault1}$) for acknowledging the start of a method. This is to be done individually for the respective function, within the Fblock Specification.

There is also a possibility to define each method in the FBlock Specification with two timing values:

1. Initial timeout between StartResult and Processing
2. A second timeout between subsequent processing messages.

All changes must be documented in the related FBlock Specification and Dynamic Specification.

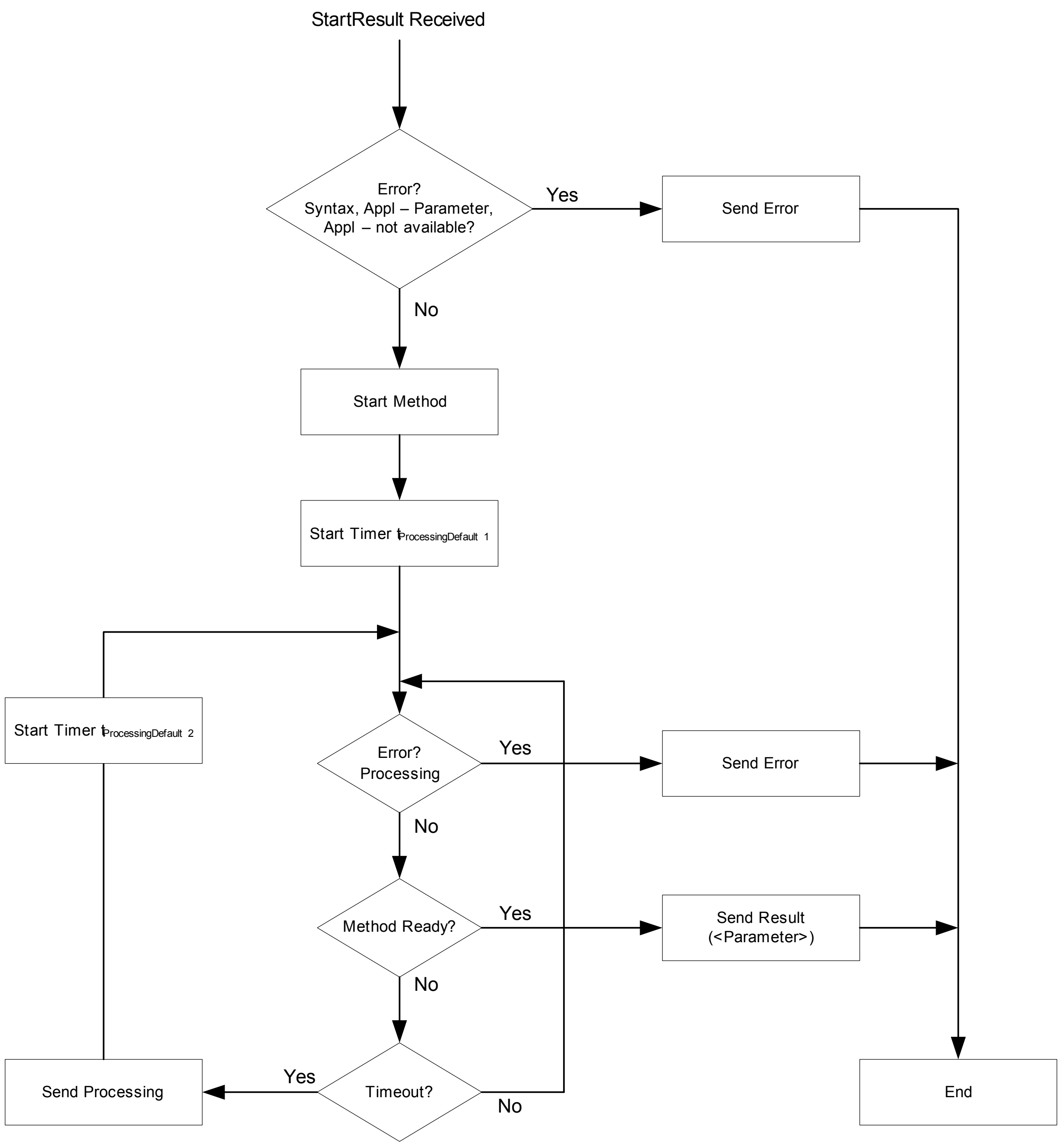

StartResult Received
Error? Syntax, Appl – Parameter, Appl – not available?
Yes
Send Error
No
Start Method
Start Timer $t_{ProcessingDefault\ 1}$
Start Timer $t_{ProcessingDefault\ 2}$
Error? Processing
Yes
Send Error
No
Method Ready?
Yes
Send Result (<Parameter>)
No
Send Processing
Yes
Timeout?
No
End

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

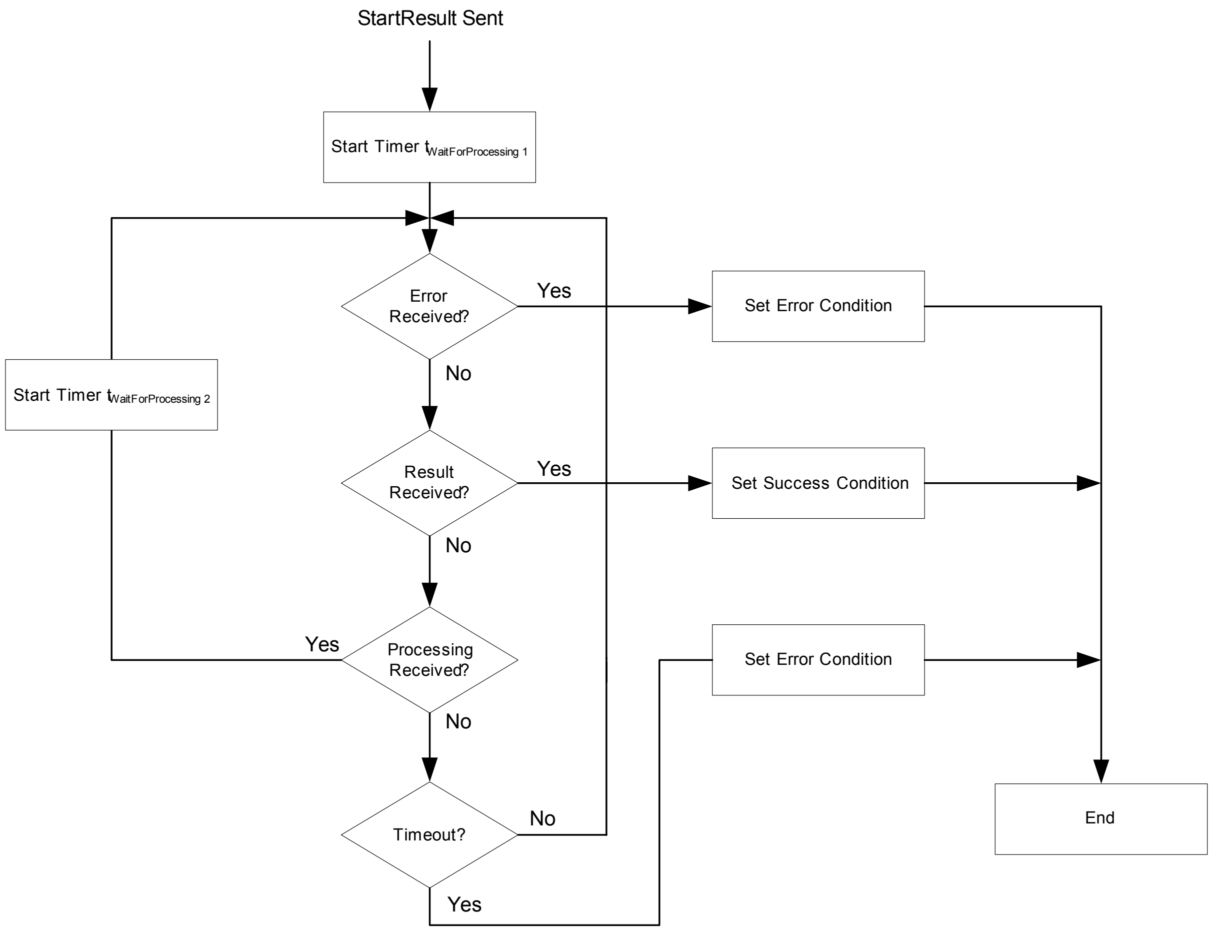


Figure 2-18: Flow for handling communication of methods (controller's side)

2.3.2.5.4 StartAck, StartResultAck, ProcessingAck, ResultAck, ErrorAck

The behavior is equal to that of Start, StartResult, Processing, Result and Error (refer to section 2.3.2.5.3 on page 50). The only difference is, that the first parameter transports the SenderHandle (refer to section 2.3.6.2 on page 71).

2.3.2.5.5 Get, Status, Error

By using OPType Get, a controller asks for the status of a property. In case of a request by using Get, a reply using Status will be 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 Get, an error can be assumed. It is not critical, if the controller reacts more tolerant and waits for a longer time. Nevertheless, an interruption of the waiting process is a must.

2.3.2.5.6 Set, Status, Error

By using Set, the content of a Property is changed. Set behaves equal to Start. 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, e.g., if the controller is registered in the Notification Matrix. In addition to that 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 function block.

2.3.2.5.7 SetGet, Status, Error

SetGet is the preferred way of controlling function blocks, for Controllers:

- that control a property only in rare cases
- which 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. This is independent of the Notification Matrix.

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 tolerant and waits for a longer time. Nevertheless, an interruption of the waiting process is a must.

2.3.2.5.8 GetInterface, Interface, Error

These OPTypes can be compared with Get, Status and Error (refer to section 2.3.2.5.5). Instead of the status, the Function Interface will be requested.

2.3.2.5.9 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) new 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.

2.3.2.5.10 Abort, Error

This OPType is 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 9) Application Error – Method Aborted: on page 49 for more information.

2.3.2.5.11 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 Error(Aborted) message. In opposite to "Abort", AbortAck transports additional "routing" information (SenderHandle, as described in section 2.3.6.2 on page 71). 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 9) Application Error – Method Aborted: on page 49 for more information.

2.3.2.6 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 Byte.

Functions that need to transport voluminous application protocols communicate via MOST High Protocol and the packet data transfer service. These functions will be marked in the FBlock Specification.

Please note:
Length is not transmitted directly via MOST, but is reconstructed from the number of received telegrams and the TelLen at the receiver's side.

2.3.2.7 Data and Basic Data Types

In principle, the data field of a message in the application layer (also referred to as Application message) may have any length up to 65535 Bytes. In a telegram on the control channel of the MOST bus, the maximum length is 12 Bytes. So longer protocols must be segmented, i.e., be sent divided up in several telegrams. It should be kept in mind that even on the application level, the data fields of a protocol should exceed 12 Bytes only in exceptional cases.

Within a data field, none, one, or multiple parameters in any combination of the following data types can be transported. They are transported 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
- Unsigned Word
- BitField
- Signed Byte
- Signed Word
- Unsigned Long
- Enum
- Stream
- Signed Long
- String
- Classified Stream
- Short Stream

Parameters are transmitted in a way that can be displayed directly. 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. 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 2.3.11 on page 77). It is not transported together with the parameter.

2.3.2.7.1 Boolean

Definition of Type	Comments
1 Byte	Only one bit can be used in each Byte.

2.3.2.7.2 BitField

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

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

2.3.2.7.3 Enum

Definition of Type	Comments
1 Byte	-

2.3.2.7.4 Unsigned Byte

Definition of Type	Comments
1 Byte	-

2.3.2.7.5 Signed Byte

Definition of Type	Comments
1 Byte	-

2.3.2.7.6 Unsigned Word

Definition of Type	Comments
2 Byte	-

2.3.2.7.7 Signed Word

Definition of Type	Comments
2 Byte	-

2.3.2.7.8 Unsigned Long

Definition of Type	Comments
4 Byte	-

2.3.2.7.9 Signed Long

Definition of Type	Comments
4 Byte	-

2.3.2.7.10 String

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

Please note:
In general, only "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	No
0x03	RDS	No
0x04	DAB Charset 0001	No
0x05	DAB Charset 0010	No
0x06	DAB Charset 0011	Yes
0x06 - 0xBF	Reserved	
0xC0 – 0xFF	Proprietary	

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	-

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

2.3.2.7.11 Stream

Definition of Type	Comments
Any Data	-

2.3.2.7.12 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 the stream.

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. If a MediaType is not available "application/octet-stream" shall be assumed when MediaType is an empty string.

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).

2.3.2.7.13 Short Stream

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

Length: 1 Byte Length of the stream (max 255 Bytes)

2.3.3 Function Formats in Documentation

The protocols have different DeviceIDs, depending on the protocol being received or transmitted. In documentation that must be human readable, the following general description must be used, which covers both cases:

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

SrcAdr and TrgAdr are the physical MOST addresses of the sending and the receiving device, respectively. On the sender's side, it is identical with the TrgAdr, and on the receiver's side with the SrcAdr (please refer to the example in section 2.3.5 on page 63). In most cases, only one instance of the function block is available in the system, and InstID can be omitted. Descriptions can also be simplified by omitting the Length.

```
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)
```

2.3.4 Protocol Catalogs

The telegrams are included in a catalog and are grouped by functions (Function catalog). It is a good approach to implement this catalog in a database, so that a printable version can be produced.

2.3.5 Application Functions on MOST Network (Introduction)

The controlling mechanisms described in this document are generally independent of the kind of bus used. Protocols 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.

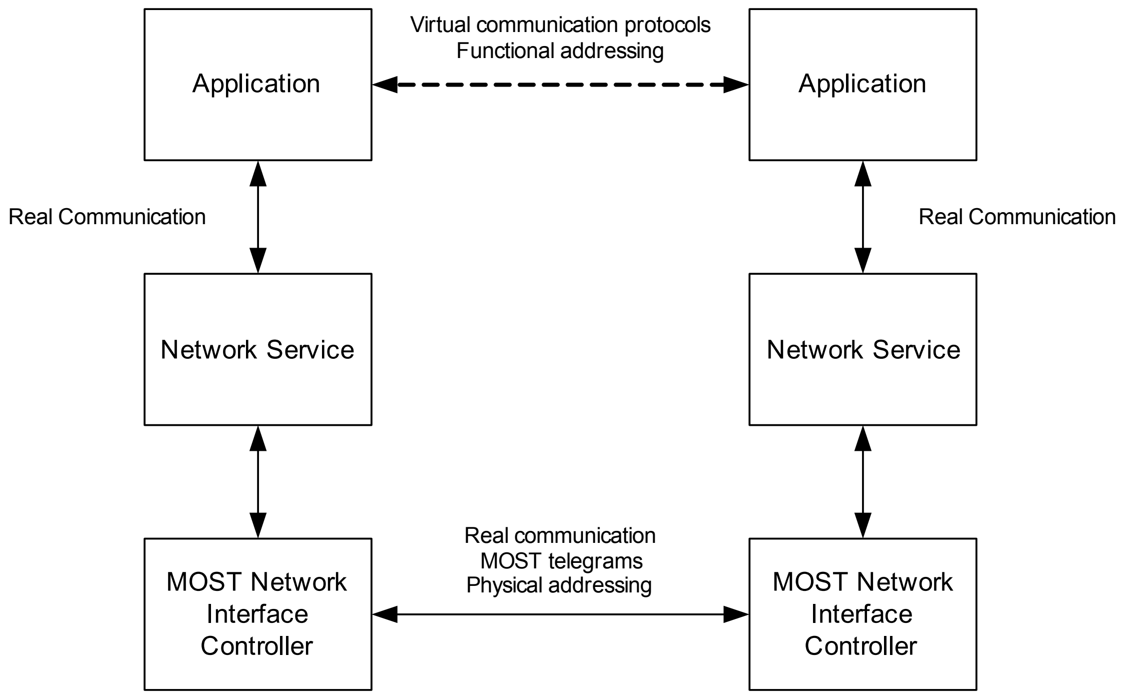


Figure 2-19: Virtual communication between two devices on application layer and real comm. via network

All application protocols are finally transferred via the control channel of the MOST Network. From the application's point of view, all protocols are passed on to the Network Service. Depending on the length, an application protocol is sent with a single transfer if it fits into one MOST telegram, otherwise via segmented transfer.

In a MOST Network, nodes, or devices, are addressed. In order to transport a protocol to a function block, the MOST telegrams are provided with the address of the device that contains the function block.

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 a function block with the FBlockID CD and the InstID 1. The function block is found in device CD Player with the physical MOST address CDC.

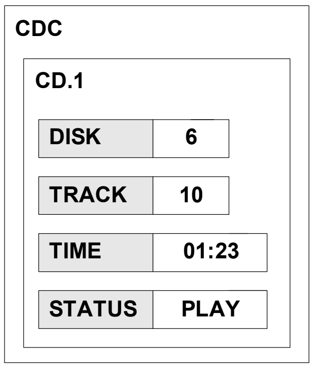


Figure 2-20: Device with MOST address CDC, a function block CD Player with FBlockID CD, and its functions

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

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

This protocol is sent by a device with the physical MOST address HMI. Therefore it will be passed on to the Network Service in the following form:

```
FFFF.CD.1.Track.Set(10)
```

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

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

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

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

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

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

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

```
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 protocol 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:

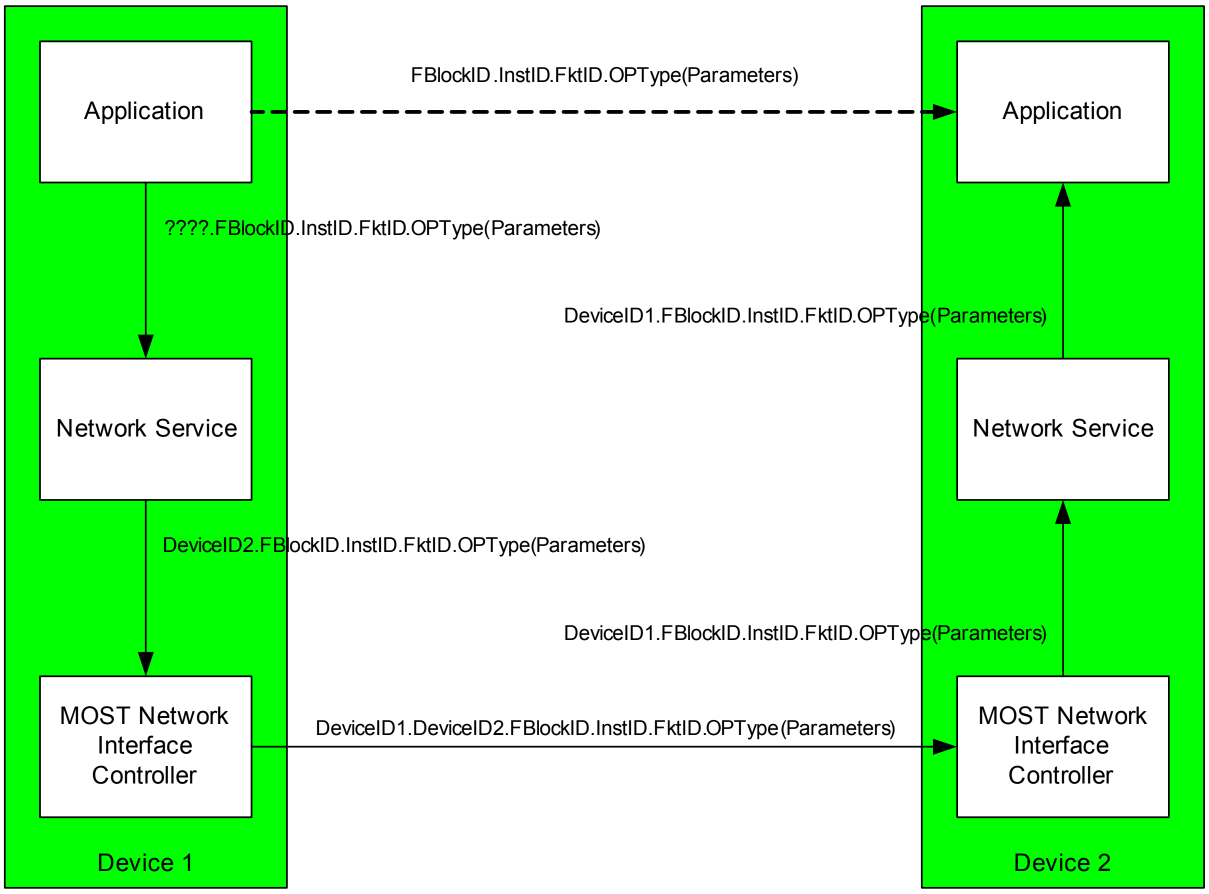

Application
FBlockID.InstID.FktID.OPType(Parameters)
Application
????.FBlockID.InstID.FktID.OPType(Parameters)
DeviceID1.FBlockID.InstID.FktID.OPType(Parameters)
Network Service
Network Service
DeviceID2.FBlockID.InstID.FktID.OPType(Parameters)
DeviceID1.FBlockID.InstID.FktID.OPType(Parameters)
MOST Network Interface Controller
DeviceID1.DeviceID2.FBlockID.InstID.FktID.OPType(Parameters)
MOST Network Interface Controller
Device 1
Device 2

Figure 2-21: Communication between two devices via the different layers

2.3.6 Controller / Slave Communication

For communication between Controllers and Slaves, properties and methods must be differentiated.

2.3.6.1 Communication with Properties Using Shadows

Below, communication between a controlling and a controlled device is explained for Properties by an example:

- Controlling device (Controller):
  Contains a function block controlling another function block.
- Controlled device (Slave):
  Contains only controlled function blocks (for demonstration purpose).

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

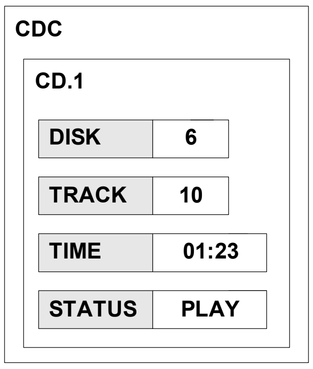


Figure 2-22: Example for a Slave device

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 protocol:

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

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

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

By sending this protocol, 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 protocol:

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

These status messages are sent by the CD player, even in a case where a property changes itself, e.g., when the player changes to the next track during play mode (on the condition that another device is registered in the notification matrix of function block CD).

The MOST device address of the CD changer (represented by the abbreviation CDC) together with FBlockID and the InstID describe the property to be changed. To make sure that the protocols for controlling a device find their way through the system, the property description must be unique in the entire system.

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

```
???->CDC1: CD.1.STATUS.SET(STOP)
???->CDC2: CD.2.STATUS.SET(STOP)
```

By this, two CD function blocks 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 (controller) contains the Shadows of the functions it controls. The Shadow of a function in the control device represents an image of the property of the Slave device. That means, for each controlled property of the Slave device, the control device contains a respective variable. For the controller, the function seems to reside in its own memory area. This is shown in the figure below:

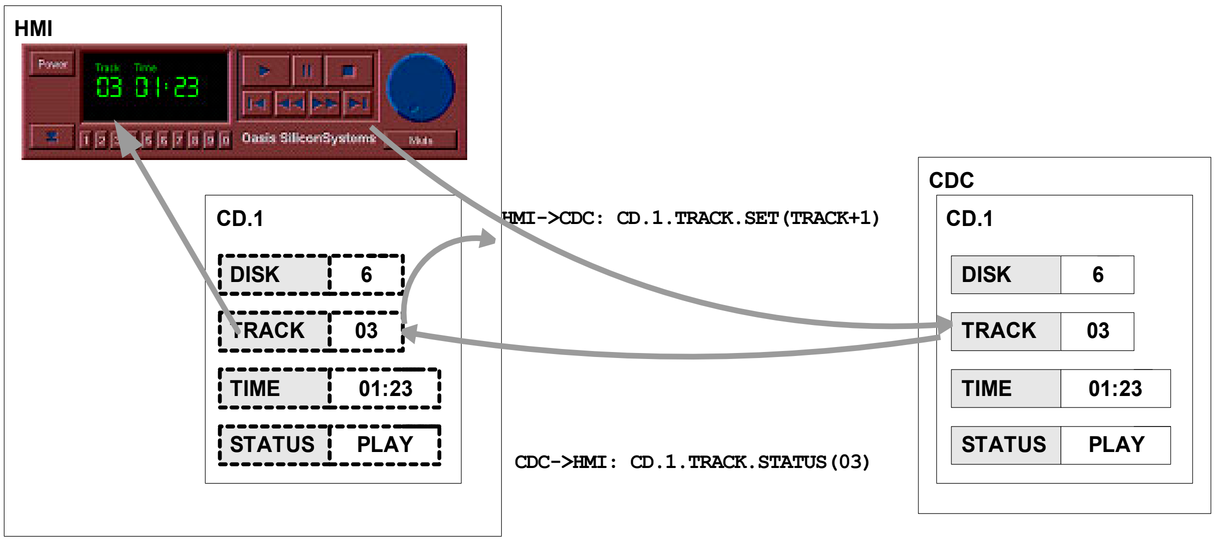


Figure 2-23: Virtual illustration of the controlled properties in the control device

The HMI shown in Figure 2-23 has an image of all properties of CDC (Slave device) represented by the variables Disc, Track, Time and Status. 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.

On a click onto the button, the HMI takes the contents of its local variable Track, increments it by one, and sends the protocol CD.1.Track.Set(Track+1) to device CDC. After the player has changed track, it replies by sending protocol CD.1.Track.Status(3). Addressing of the response is equal to the addressing of the command, except the address, since the answer is sent to a (virtual) identical function block. Variable Track reacts only on that protocol and stores the new value. The change of variable Track causes the HMI to update its display.

As shown in the figure below, there is one protocol assigned to each variable unambiguously. Every variable in HMI “reacts” only on the assigned protocol, sent from the respective device.

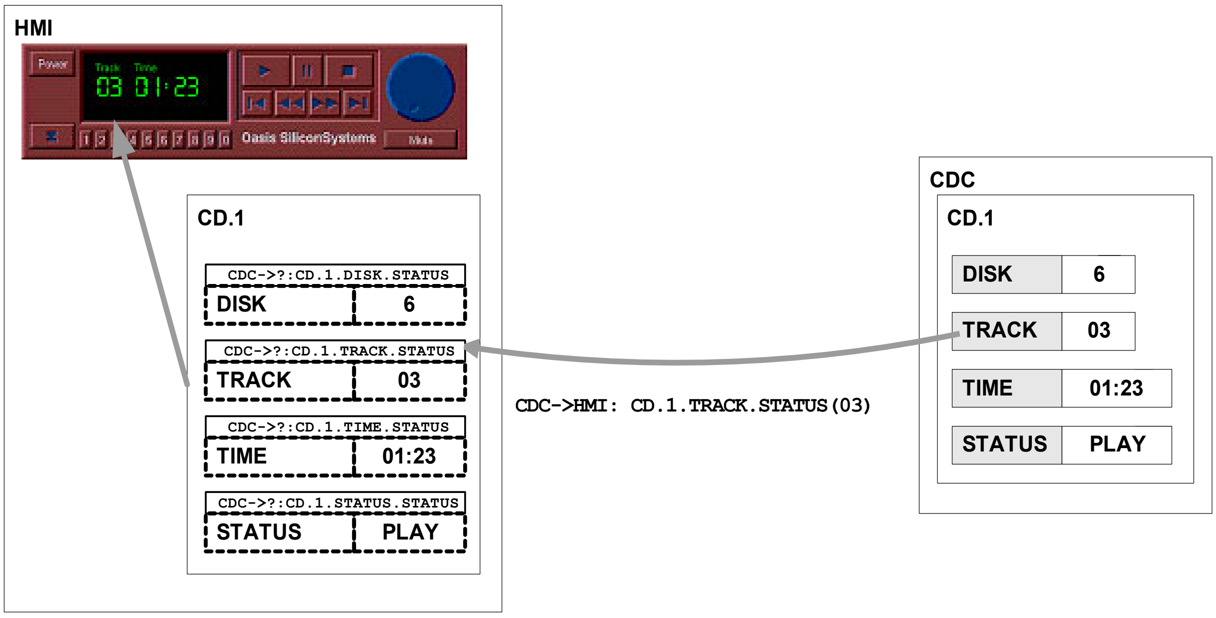


Figure 2-24: Unambiguous assignment between protocol and variable

The figure below shows the advantage of 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.

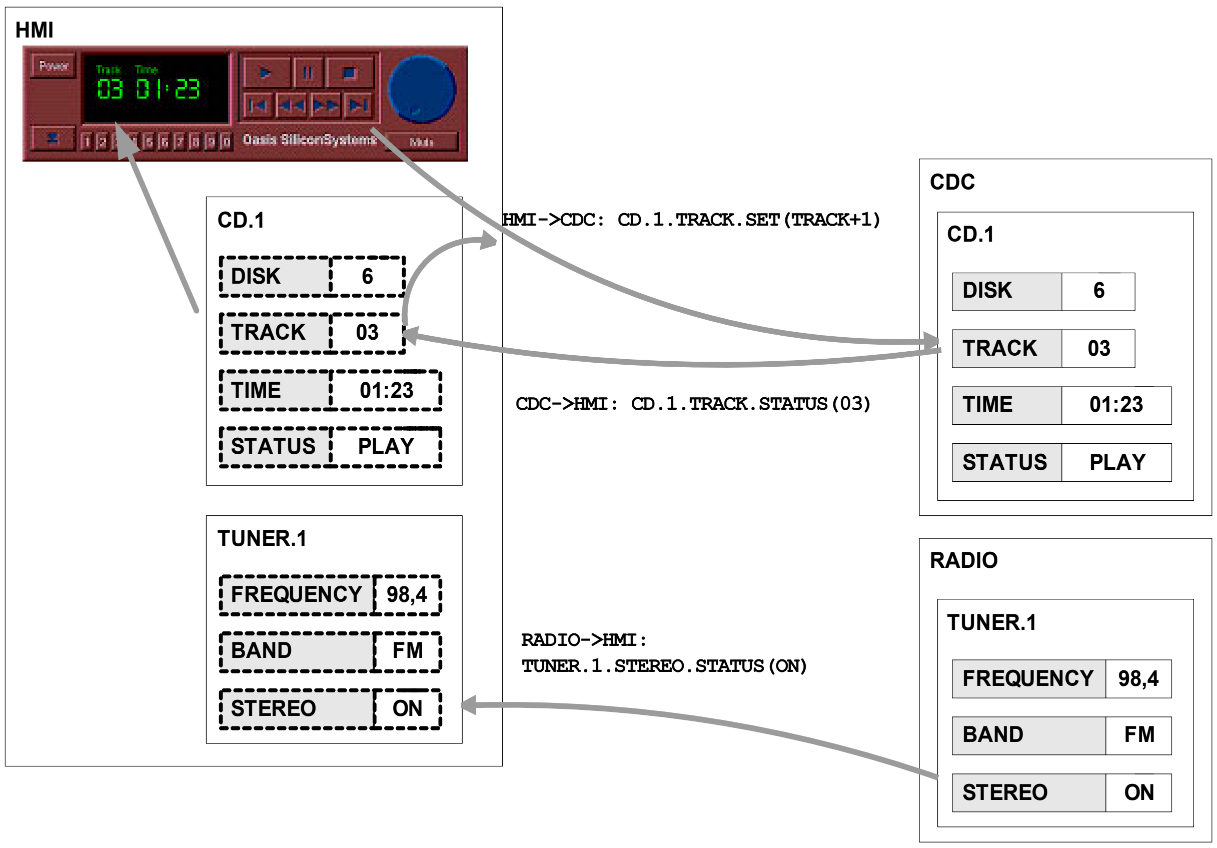


Figure 2-25: Controlling multiple devices

A similar case could be imagined, if several identical CD players are available in the network. Operation mode of the HMI could be changeable, for example, between CD1 and CD2. The display would show only the status of the currently selected player, and the keyboard would be switched, too.

As shown in the graphic below, such a HMI would contain two sets of variables (shadows), one for each CD player. The variables for CD.1 react only upon protocols of CD.1, while the variables for CD.2 react only upon protocols of CD.2. If both of the function blocks 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.

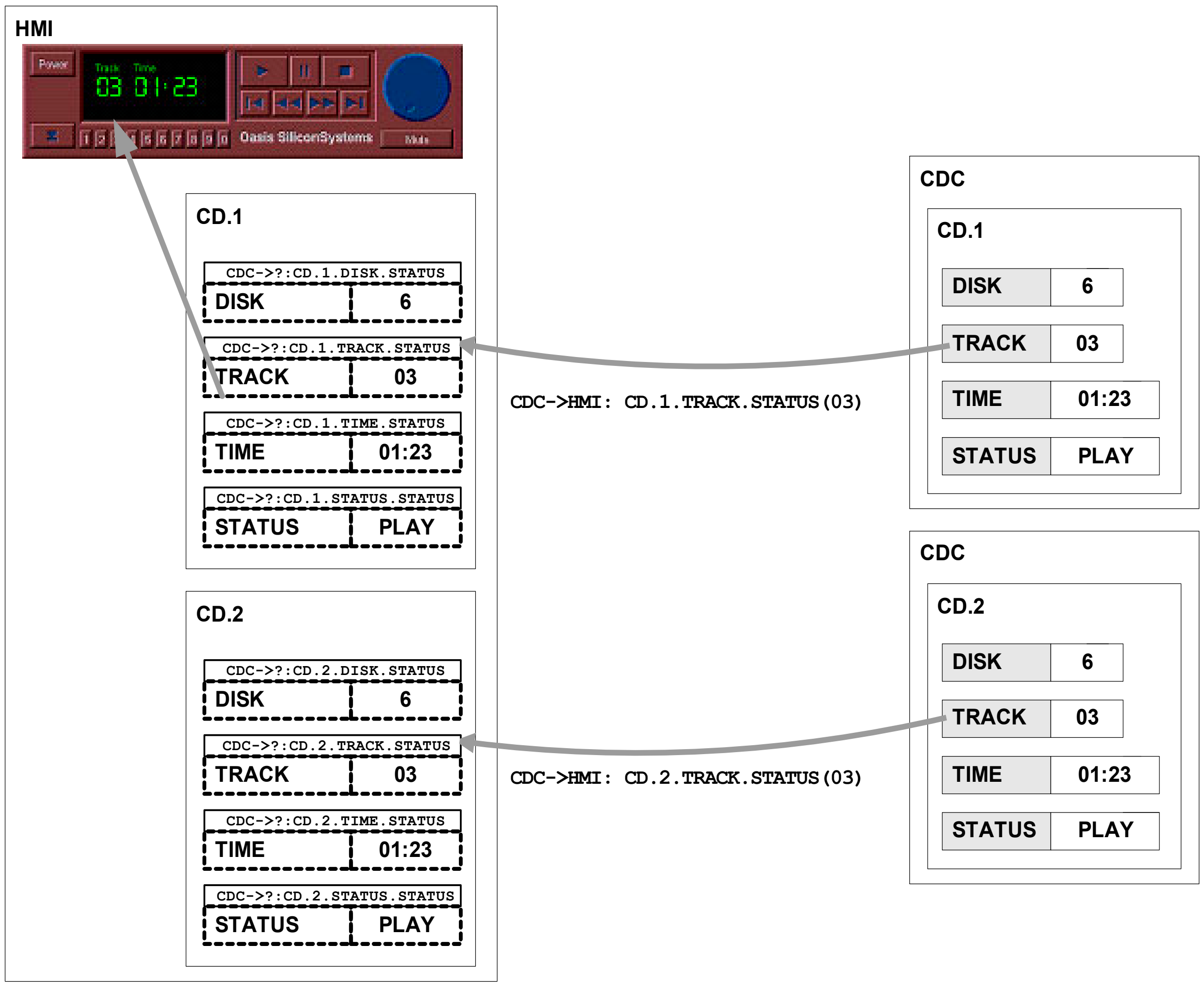


Figure 2-26: Controlling two identical devices

For the assignment of protocols and variables in the control device, the respective protocols are defined for each variable. Each variable therefore has a filter function that can be passed only by the "own" protocol.

This can be done by a table, which contains the protocols consisting of MOST sender address, FBlockID, InstID, and FktID. There can be one pointer assigned to each protocol, pointing to the respective variable. In addition to that, or as an alternative, function pointers are also allowed. By this, functions could be called depending on protocols, and controlled by tables.

The concept can also be realized by an object-oriented approach, where variables are realized by objects with protocol filters and methods for representation. Following this approach, all incoming protocols are distributed to all objects, but only that object whose filter lets the protocol pass, will react. Analogously, the incoming protocols are compared to all protocols in the table when using the table approach.

On more complex control devices, this approach can be optimized by filtering the protocols step by step. The figure below shows an HMI, which contains Shadows of a CD player and a tuner. These Shadows are implemented as interface objects. The interface objects are combined in two parent objects that filter the incoming protocols by sender address and InstID. The interfaces themselves only need a filter for the FktID.

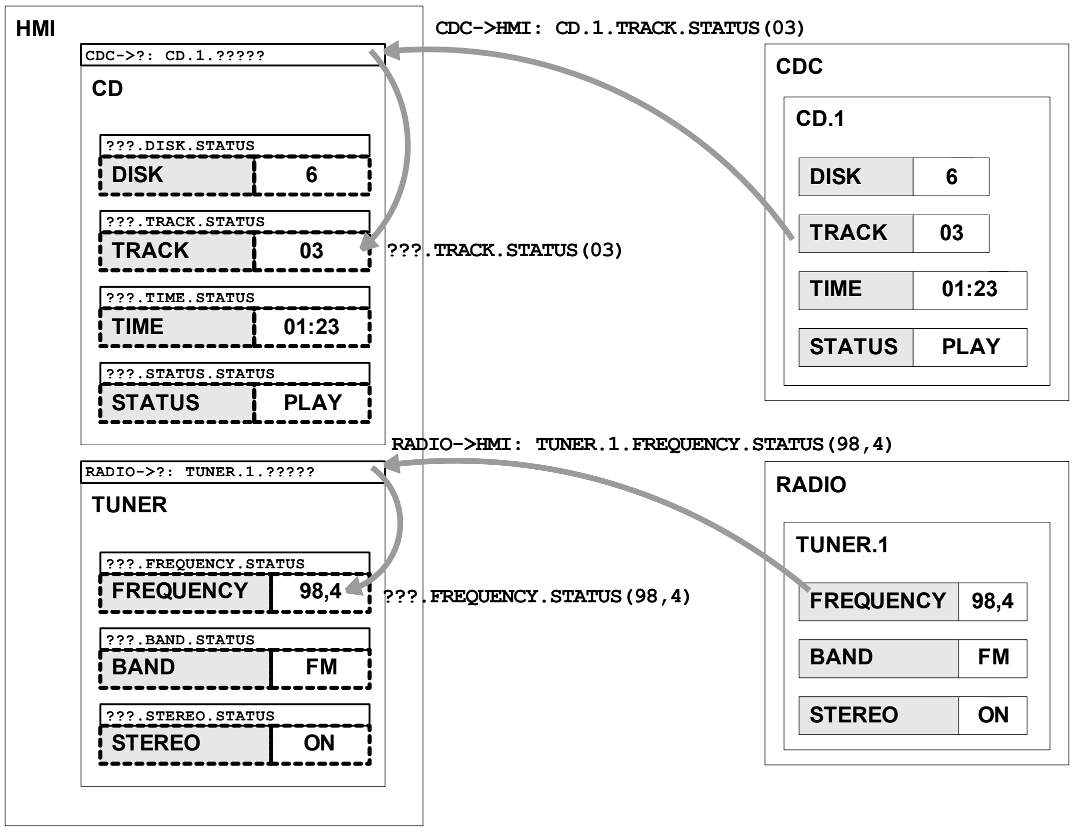


Figure 2-27: Hierarchical structure of the protocol filter (command interpreter)

Without object-oriented programming, the stepwise filtering can be implemented by using a message dispatcher. This dispatcher would forward the protocols to the respective function blocks based on sender address, FBlockID and InstID. Every function block can then analyze the FktIDs itself, by an own command interpreter.

Based on the well-structured protocols, further analyzing steps can be inserted if required.

2.3.6.2 Communication with Methods

2.3.6.2.1 Standard Case

In general, communication with properties is equal to communication with methods. This means that a controller controls a function in a Slave Device and there will be a reply to the Controller Device. An example:

```
Controller -> Slave: FBlockID.InstID.StartResult  (Data)

Slave -> Controller: FBlockID.InstID.Result  (Data)

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

2.3.6.2.2 Special Case Using Routing

In some cases there are methods where the general way of communication is not sufficient. The philosophy of building Shadows when on handling properties is based on the fact, that every property has only one single and unique state. This state then is imaged on one or more controllers. This condition is not valid for methods. So it may happen that a method is processing a request for one Controller, while it appears to another Controller to be busy. It has many states.

In addition to that, in methods a process is triggered, which has a longer processing time. The Controller may need to wait for a result. If several tasks within a device accessed one method at the same time, it must be possible to route the answer back to the respective task.

One example can be the SMS service in a GSM module of the device Telephone. In HMI, three tasks desired to send an SMS message independently from each other. The message of task 1 was sent, the one of task 2 was buffered, while the message of task 3 was rejected. The respective status message must now be assigned, which is not possible using the communication methods described up to now.

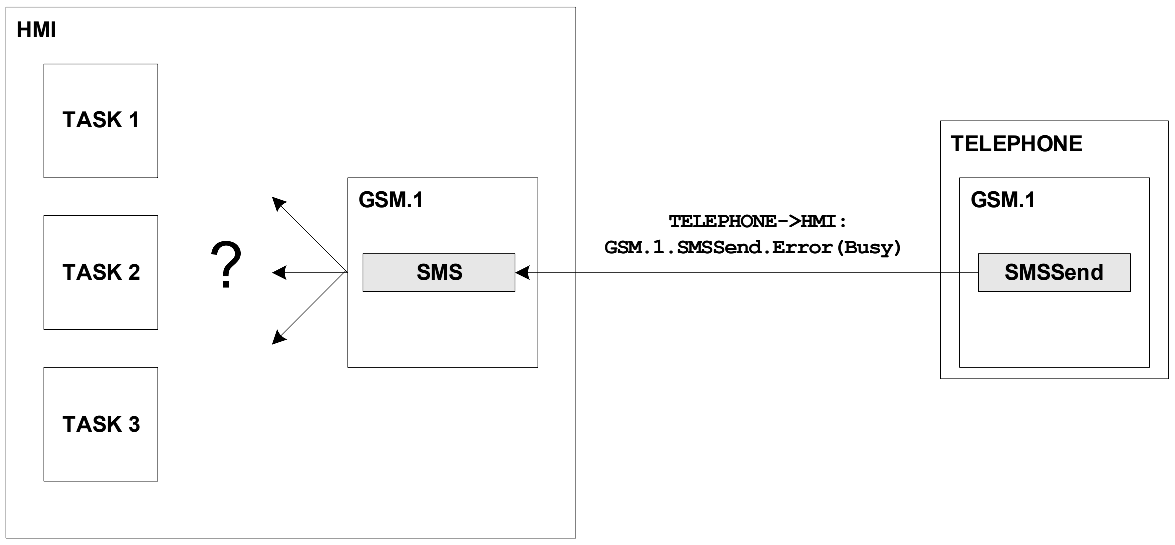


Figure 2-28: Routing answers in case of multiple tasks (in one controller) using one function

To provide routing in such cases, the OPTypes StartResultAck, ProcessingAck, ResultAck, and ErrorAck are introduced. The behavior of these OPTypes is identical to that of StartResult, Processing, Result and Error. The only difference is that as first parameter the SenderHandle (data type unsigned word) is inserted. The SenderHandle is set by the Controller at StartResultAck and characterizes the sender more in detail (Task, process...). The SenderHandle will not be interpreted by the Slave, but will be returned in an answer (ProcessingAck, ResultAck or ErrorAck).

The SMS call in Task 1 may look like:

```
Controller -> Slave: Telephone.1.SMSSend.StartResultAck
(SenderHandle1.SMSData)
```

After successful transmission, Task 1 gets:

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

If Task 3 desires to send in the meantime, it sends:

```
Controller -> Slave: Telephone.1.SMSSend.StartResultAck (SenderHandle3.SMSData)
```

And it then gets in return:

```
Slave -> Controller: Telephone.1.SMSSend.ErrorAck (SenderHandle3.ErrorCode=”Busy”)
```

It must be decided individually, which methods must have a detailed back addressing with OPTypes StartResultAck, ProcessingAck, ResultAck, and ErrorAck.

2.3.7 Seeking Communication Partner

It may happen that an application has to seek a communication partner, that is, a function block. This may happen in a self-configuring audio system with four or six active speakers. The audio controller knows that function blocks with the FBlockID AudioAmplifier must be available, but does not know how many, or where. Therefore it has to seek, and gets the instance IDs as reply. With the help of the InstIDs and the number of audio amplifiers, it can configure itself correctly.

To seek a function block, the seeking block sends the following protocol to the NetworkMaster:

```
control -> ??? : 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 physical and functional address:

```
??? -> control : NetworkMaster.CentralRegistry.Status (
                 Rx/TxLog.FBlockID.InstID,
                 Rx/TxLog.FBlockID.InstID,...)
```

Optionally, the InstID can also be specified, to search for a certain function block:

```
control -> ??? : NetworkMaster.CentralRegistry.Get ( FBlockID.InstID )
```

If the respective function block does not exist, the NetworkMaster replies with an error and error code 0x07 “Parameter not available”. It returns the number of the parameter (0x01 in this case) and the value (FBlockID.InstID in this case).

2.3.8 Requesting Function Block Information from a Device

To obtain information about the function blocks contained by a device, every NetBlock has the property **FBlockIDs** (0x000). It will be read in the following way:

```
control -> ??? : NetBlock.FBlockIDs.Get
```

and answers with a list of the contained FBlockIDs. The function block that most characterizes the device (e.g., Tuner in a radio device) is listed first. The NetBlock and function block EnhancedTestability do not need to be listed, as they are mandatory function blocks in every device:

```
??? -> control : NetBlock.FBlockIDs.Status (FBlockID1.InstID1,
                                            FBlockID2.InstID2...
                                            FBlockIDN.InstIDN)
```

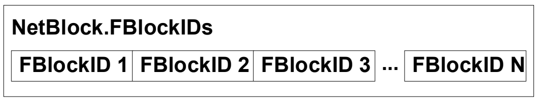


Figure 2-29: Reading the function blocks of a device from NetBlock

2.3.9 Requesting Functions from a Function Block

In an adaptable system it may happen that a controller does not know exactly which functions are available in a function block (e.g., simple or high-end audio amplifier). Therefore, every function block has the function **FktIDs** (0x000). It is read as follows:

```
    control -> ??? : FBlockID.InstID.FktIDs.Get
```

Within a function block, FktIDs between 0x000 and 0xFFF (4096 different FktIDs) can be available. The FktIDs are assigned as described in 2.3.2 on page 38. This raises the problem of a compact response, if the functions contained in a function block are requested. It is solved by a mechanism derived from the run length encoding. 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:

```
    ??? -> control: FBlockID.InstID.FktIDs.Status (002 004 006 022 024 A00 A02 0)
```

The last 0 represents a stuffing nibble.

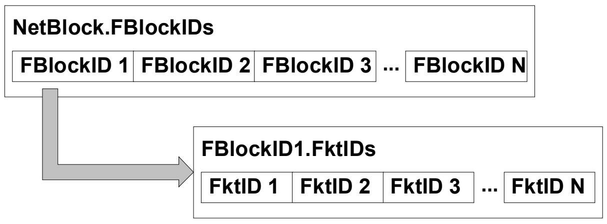


Figure 2-30: Requesting the functions contained in an application block

2.3.10 Transmitting the Function Interface

2.3.10.1 Principle

In principle, function interfaces can be transmitted to a controller, or a HMI.

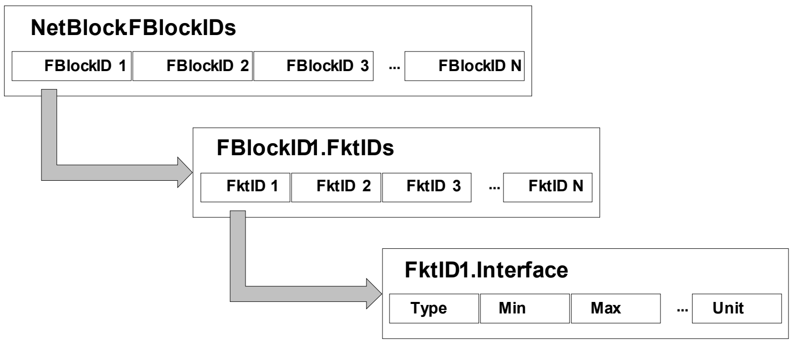


Figure 2-31: Requesting the function interface of a function

The flow for determining all function interfaces of a function block looks like:

```
control -> slave : FBlockID1.FktID1.GetInterface
slave -> control : FBlockID1.FktID1.Interface ( [ Interface Description ] )
control -> slave : FBlockID1.FktID2.GetInterface
slave -> control : FBlockID1.FktID2.Interface ( [ Interface Description ] )
...
control -> slave : FBlockID1.FktIDN.GetInterface
slave -> control : FBlockID1.FktIDN.Interface ( [ Interface Description ] )
```

The parameter list “Interface Description” contains information about a function interface.

2.3.10.2 Realization of the Ability to Extract the Function Interface

In the FBlock Specifications, every interface of classified functions is described. By doing this, a classified definition of application protocols, as well as a uniform description, is possible, which can be based onto a few classes, which are described in the section 2.3.11.

2.3.11 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.

2.3.11.1 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 2.3.2.7 on page 56) 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 a variable of type Classified Stream.

Table 2-7: Classes of functions with a single parameter

The function classes (basic classes) with one variable and their resulting protocols are described in detail below. The following universal parameters are used:

Flags: 8 Bit

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 (like the gray options in PC application's menus). This is done by setting the Enabled bit to 0. Bit 2 indicates whether a function uses Unicode or standard strings (note: Unicode is not ASCII compatible). 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 2-8 shows the three possible modes.

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

Table 2-8: The different modes of the bit field Channel Type

The meaning of the characters "C" and "A" in the table is as follows:
C: messages on control channel without using MOST High
A: messages on the asynchronous channel using MOST High

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

Mode 1:
All communication is done via the MOST High Protocol on the asynchronous channel. The only exceptions from this are the OPTypes GetInterface and Interface which 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.

Mode 2:
This is a mixed mode where only the OPTypes that are carrying a lot of data are accessed over the asynchronous channel via the MOST High Protocol. An exception is processing which does not contain a lot of data but is sent in the same way as Result i.e., over the asynchronous channel.

Bit 6 and 7 are reserved for future use.

Class:	8 Bit	0x00	Unclassified method	
		0x10	Unclassified property	
		0x11	Switch	
		0x12	Number	
		0x13	Text	
		0x14	Enumeration	
		0x15	Array	Refer to section 2.3.11.2.2 on page 92.
		0x16	Record	Refer to section 2.3.11.2.1 on page 90.
		0x17	Dynamic Array	Refer to section 2.3.11.2.3 on page 95.
		0x18	Long Array	Refer to section 2.3.11.2.4 on page 97.
		0x19	BoolField	
		0x1A	BitSet	
		0x1B	Container	
		0xFF	Abort	(No further specifications behind this location)

OPTypes: 16 Bit

BitField 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.

2.3.11.1.1 Function Class Switch

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

Boolean:	1 Byte	0 for off, 1 for on

Example: RDSOnOff in AM/FMTuner1

```
Function:     RDSOnOff                    e.g., 0x00A

Flags:        visible, enabled,           0000 1011 = 0x0B
              no Unicode, notification

Class:        Switch                      0x11

OPTypes:      Get, SetGet, Status         1101 0000 0010 0110 = 0xD026
              GetInterface, Interface,
              Error

Name:         RDSOnOff                    "RDS"
```

Upload interface:

```
Tuner -> HMI: AM/FMTuner.1.RDSOnOff.Interface (0B 11 D026 "RDS")
```

Setting RDS = OFF:

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

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

2.3.11.1.2 Function Class Number

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

DataType:	Uns. 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:	Uns. 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 like the Number parameter.
Units:	Uns. Byte	Unit

Unit	Encoding
none	0x00
Distance:	
cm	0x01
m	0x02
km	0x03
miles	0x04
Time:	
us (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
Consumption:	
l/100km	0x40
miles/gal	0x41
km/l	0x42

Unit	Encoding
Speed:	
km/h	0x50
Miles/h	0x51
m/s	0x52
Temperature:	
°C	0x60
F	0x61
Volume:	
dB	0x70
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

Table 2-9: Available units

2.3.11.1.3 Function Class Text

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

MaxSize:	Uns. Byte	Maximum length of string

2.3.11.1.4 Function Class Enumeration

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

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

Please Note:
Increment and Decrement must be interpreted like Predecessor and Successor in common programming languages.

2.3.11.1.5 Function Class BoolField

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

Content:	Uns. Byte Uns. Word Uns. Long	Data area, containing e.g., flags
NElements:	Uns. Byte	Number of Elements in the BoolField
BitName:	String	Null terminated string, indicating the name of the respective element
BitSize:	Uns. Byte	Number of Bits required for encoding the element. Encoding starts at the LSB.

If a variable of Class BoolField is defined, a field of either 8bits, 16bits, or 32bits will be reserved. Using the flags starts at the LSB. The value 0b**** ***0 means false and 0b**** ***1 means true. Manipulating a BoolField always requires the writing of the entire variable.

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

2.3.11.1.6 Function Class BitSet

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

Size:	Uns. Byte	Size of SetOfBits (Mask + Data) in Bytes
SetOfBits:	BitField	

BitSet in Arrays and Records:

A BitSet represents one variable. That means it is addressable as an entity via one dedicated value of Pos.

Example:

```
MyArray = Array of BitSet:    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,1000 1001
                              XXXX XXXX,1110 0011
                              XXXX XXXX,0010 1100
                              XXXX XXXX,1000 1110
```

2.3.11.1.7 Function Class Container

OPType	Parameter
Set	Classified Stream
Get	Classified Stream
Status	Classified Stream
SetGet	Classified Stream
GetInterface	
Interface	Flags, Class, OPTypes, Name, MaxLength
Error	ErrorCode, ErrorInfo

The Function Class Container is used for objects that can't be described in a satisfying way by the other structures.

MaxLength:	Unsigned Word	MaxLength indicates the max size of the stream in Bytes.

2.3.11.2 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 whose dimension exceeds two (array of array of record, or a record with two arrays) are definitely not allowed. In addition to that, 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.
Dynamic Array	Properties of this class use a more dynamic approach than ordinary Arrays.
Long Array	Properties of this class are used to handle large arrays in a sophisticated way.
Sequence Property	Properties of this class contain a number of single properties of the same kind.

Table 2-10: Classes of functions with a multiple parameters.

2.3.11.2.1 Function Class Record

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

In the interface description of a record, the OPTypes are omitted, since they are not necessarily identical for all parameters. OPTypes are therefore relevant only in basic types.

NElements	Uns. 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. In case of OPTypes (internal OPTypes here) only Set, Get, Status, Increment, Decrement and Error can be used.

Please 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, ...**
BitSet	Class, OPTypes, Name, **Size**
Array	Class, Name, **NElements, IntDesc**

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 line 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 a “line” 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.

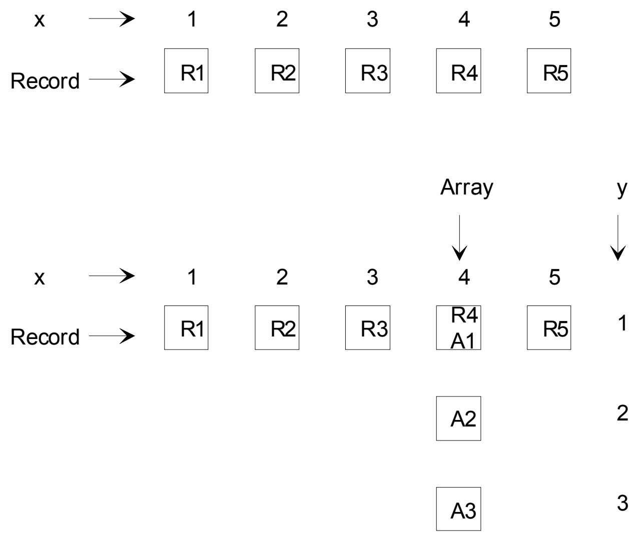


Figure 2-32: Meaning of position x in record (above) and of position y in a record with array (below)

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

2.3.11.2.2 Function Class Array

OPType	**Parameters**
Set	Position, Data
Get	Position
Status	Position, Data
SetGet	Position, Data
Increment	Position, NSteps
Decrement	Position, NSteps
GetInterface	
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...**

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

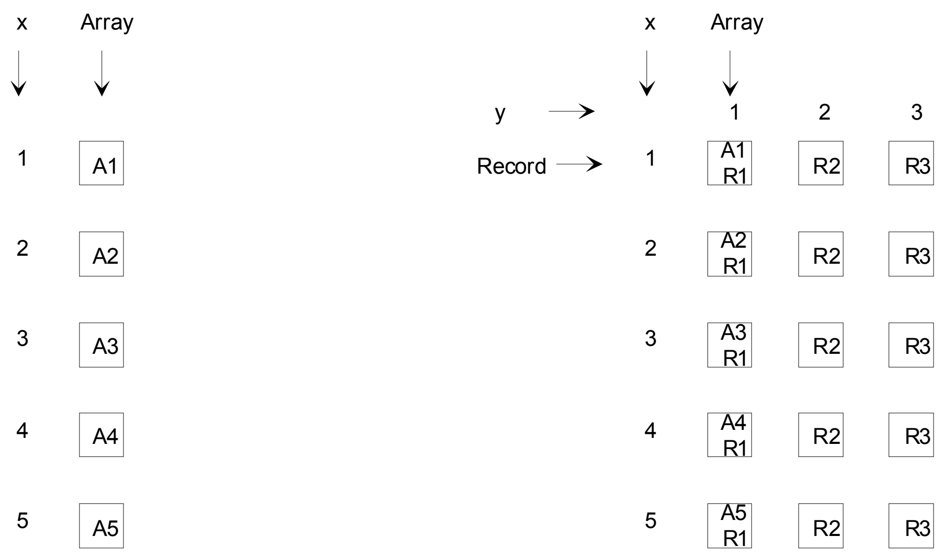


Figure 2-33: Position x in case of an array of basic type (left), y in case of an array of record (right)

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.

Please note:
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 take care to 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 (NMax entry in the function interface FI). 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, an error is assumed. 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. When requesting the FI, the formal answer is:

```
AudioDiskPlayer.0.Magazine.Interface

        (Flags. Class. Name. NMax.                                       | Array of
        Class. Name. NElements.                                          | Record of
        Class. OPTypes. Name. MaxSize                                    | Text
        Class. OPTypes. Name. Units, DataType, Exponent, Min, Max, Step  | Number
        Class. OPTypes. Name. Units, DataType, Exponent, Min, Max, Step) | Number
```

or more related to the contents:

```
AudioDiskPlayer.0.Magazine.Interface

        (Flags. Array. "Magazine", 0A
        Record. "DiskInfo", 03
        Text. OPTypes. "DiskTitle", FF
        Number. OPTypes. "TotalTime". Seconds. Word. 00. 00 00. FF FF. 00 01
        Number. OPTypes. "Tracks". 00. Unsigned Byte. 00. 01. 63. 01)
```

On request

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

one receives the title of the third disk. On request

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

the titles of all disks are returned.

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

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

On requesting the FI, the formal answer is:

```
     AudioDiskPlayer.0.Disk.Interface

          (Flags. Class. Name. NMax.                                           | Array of
          Class. Name. NElements.                                              | Record of
          Class. OPTypes. Name. MaxSize                                        | Text
          Class. OPTypes. Name. Units, DataType, Exponent, Min, Max, Step)     | Number
```

or more related to the contents:

```
     AudioDiskPlayer.0.Magazine.Interface

          (Flags. Array. "Disk", 63
          Record. "TrackInfo", 02
          Text. OPTypes. "TrackTitle", FF
          Number. OPTypes. "TotalTime". Seconds. Word. 00. 00 00. FF FF. 00 01
```

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 e.g., 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).

2.3.11.2.3 Function Class Dynamic Array

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 Dynamic Arrays this is not possible, since here, lines can be inserted or removed (the sequence may vary). So a special function class DynamicArray is introduced, where PosX will be replaced by a uniquely defined handle, the Tag of data type Unsigned Word. It is defined as first parameter in the record:

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

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

OPType	**Parameter**
Set	Tag, PosY, Data
Get	Tag, PosY
Status	Tag, PosY, Data
SetGet	Tag, PosY, Data
Increment	Tag, PosY, NSteps
Decrement	Tag, PosY, NSteps
GetInterface	
Interface	Refer to section 2.3.11.2.2 on page 92
Error	ErrorCode, ErrorInfo

Tag	uns. Word	=	0x00 00	all lines
		<>	0x00 00	one special line
PosY	uns. Byte	<>	0x00	one special column (only if Tag <> 0x00 00)
		=	0x01	not allowed, no access to Tag

Please note:
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 Dynamic Array 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 = 0x00 00 and PosY = 0x00
3. Tag = 0x20 06 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:

Like in case of 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), e.g., several telephone directories. So there is no need for an individual instance per array. So these functions will be placed in the range of Coordination (0x000..0x1FF).

By DynArrayIns (FktID=0x080), a number Quantity (Uns. Word) of array elements (entire lines) will be inserted in DynamicArray FktID. The lines will be inserted after that line containing Tag. The data contents of the lines to be inserted will be transferred as Data.

```
DynArrayIns.Start (FktID, Tag, Quantity, Data)
```

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

```
DynArrayDel (FktID, Tag, Quantity)
```

Examples:

DynArrayDel (FktID, 00 00, FF FF)	Deleting of entire array
DynArrayDel (FktID, 87 95, FF FF)	Deleting of entire array starting at line containing Tag 0x8795
DynArrayDel (FktID, 87 95, 00 01)	Deleting of the line containing Tag 0x8795
DynArrayDel (FktID, 87 95, 00 00)	No deleting

2.3.11.2.4 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 a function class ArrayWindow. It is possible to generate instances of class ArrayWindow dynamically. An ArrayWindow represents an extract, a window to the MotherArray. An instance of LongArray therefore consists of at minimum two functions, so LongArray is no simple function class.

2.3.11.2.4.1 MotherArray

The MotherArray is structured like a function of class DynamicArray. So communication of DynamicArray is identical to the communication of MotherArray, but there are only the OPTypes GetInterface, Interface and Error available (refer to section 2.3.11.2.2 on page 92).

The main difference compared to DynamicArray is, that the MotherArray is not controlled and viewed directly, but via one or more different functions. In the function interface of the MotherArray, all OPTypes are listed that can be executed via ArrayWindows. Below there is an example for 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

2.3.11.2.4.2 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. So there must be an individual ArrayWindow for each Controller. So it is possible that several HMIs have individual ArrayWindows to an address directory (MotherArray), which have different size and position.

So functions of class ArrayWindow are instantiated dynamically at runtime. Therefore a function block (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 FktID 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

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	uns. Word	Top left corner of the ArrayWindow is positioned at PositionTag
WindowSize	uns. Byte	Number of elements contained by the ArrayWindow

The methods CreateArrayWindow and DestroyArrayWindow can instantiate and destroy ArrayWindows even of several MotherArrays. If e.g., 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 e.g., three telephone directories may be watched by three ArrayWindows.

If a device enters sleep mode, all instances of ArrayWindows are destroyed. Every Controller stores the position of its ArrayWindow with the help of the Tag of the first line. During CreateArrayWindow, and by the help of Move (FktIDArrayWindow, Absolute, Tag), the window can be positioned again.

The status of an ArrayWindow is kept up to date in the Controller by using a shadow. Also in that case, it is the Slave's task to keep the shadow up to date. 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.

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

MotherArray

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

ArrayWindow

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

SetGet (9473, 03, B3)

There is no function interface for an ArrayWindow, since it only represents a "view" onto the MotherArray. The MotherArray itself has a function interface that describes all operations that can be performed by using an ArrayWindow.

Function	OPType	Parameter
ArrayWindow	Set	Tag, PosY, Data
	Get	Tag, PosY
	Status	Tag, PosY, CurrentSize, AbsolutPosition, Data
	SetGet	Tag, PosY, Data
	Increment	Tag, PosY, NSteps
	Decrement	Tag, PosY, NSteps
	GetInterface	
	Interface	Refer to section 2.3.11.2.2 on page 92
	Error	ErrorCode, ErrorInfo

Tag	Uns. Word	=	0x00 00	all lines
		<>	0x00 00	one special line
PosY	Uns. Byte	<>	0x00	one special column (only if Tag <> 0x00 00)
		=	0x01	not allowed, no access to Tag
CurrentSize	Uns. Word	<>	current size of the MotherArray	
AbsolutPosition	Uns. Word	<>	absolute position of the Array Window in the Mother Array. The value specifies the position of the top left cell in the MotherArray and the counting starts at 0.)	

2.3.11.2.4.3 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 MoveAW is mandatory. An instance of MoveAW is used for all instances of ArrayWindows (FktID) of a function block.

```
MoveAW.Start (FktID, Mode, Number, Tag}
```

FktID		FktID of the ArrayWindow to be moved	
Mode	uns. Byte	00	Top
		01	Bottom
		02	Up
		03	Down
		04	Absolute

Top and Bottom:

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

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
	.	
	.	
9643	w	1D
6354	x	1E
3425	y	1F
1045	z	20
FFF	FF	FF

MotherArray

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

ArrayWindow

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

MoveAW.Start (FktID, Top, xx, xxxx)

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

MoveAW.Start (FktID, Bottom, xx, xxxx)

Up and Down:

Up and Down are used for relative movement of the ArrayWindow, where the parameter Number (Uns. 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.

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

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

MoveAW.Start (FktID, Up, 03, xxxx)

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

MoveAW.Start (FktID, Down, 05, xxxx)

Absolute:

Absolute adjusts an ArrayWindow in a way, that the first line contains the desired Tag. If the Tag located too far by the end of the MotherArray, so that the ArrayWindow would exceed the valid range, the ArrayWindow will be placed like in case of using 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
FFF	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

MoveAW.Start (FktID, Absolute, xx, 2100)

The second method SearchAW is optional. SearchAW provides a seeking of Searchstring in MotherArray through ArrayWindow (FktID). Search is performed in that element of each line, which is specified by PosY:

```
SearchAW.Start (FktID, 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").

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

MotherArray

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

ArrayWindow

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

SearchAW.Start (FktID, 02, „c“)

2.3.11.2.4.4 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 exists. It works like a normal array except that it is only possible to do Get on it.

2.3.11.2.5 Function Class Sequence Property

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

NElements	Uns. 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 here in case of elements, parameter Flags is not available. In case of OPTypes (internal OPTypes here) only Set, Get, Status, Increment, Decrement and Error can be used.

2.3.11.3 Function Classes for Methods

For methods there are only two function classes, since methods may differ significantly with respect to the parameters transferred during Start and Result (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 kind of method is used to trigger something. They have no parameters.
Sequence Method	This kind of method has a number of parameters, all of the same kind.
Unclassified Method	Methods that do not belong to any classified function class belong here.

Table 2-11: Classes of functions for a method.

2.3.11.3.1 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	Parameter
Start	
Processing	
Result	
StartResult	
StartResultAck	SenderHandle
ProcessingAck	SenderHandle
ResultAck	SenderHandle
ErrorAck	SenderHandle, ErrorCode, ErrorInfo
GetInterface	
Interface	Flags, Class, OPTypes, Name
Error	ErrorCode, ErrorInfo

2.3.11.3.2 Function Class Sequence Method

OPType	Parameter
Start	<Parameter>{, <Parameter>}
Processing	
Result	<Parameter>{, <Parameter>}
StartResult	<Parameter>{, <Parameter>}
StartResultAck	<Parameter>{, <Parameter>}
ProcessingAck	
ResultAck	<Parameter>{, <Parameter>}
Abort	
GetInterface	
Interface	Flags, Class, Name, Nelements, IntDesc1, InDesc2, …
ErrorAck	ErrorCode, ErrorInfo
Error	ErrorCode, ErrorInfo

NElements	Uns. 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 here in case of elements, parameter Flags is not available.

2.3.12 Handling Message Notification

In many cases, HMIs and controllers must get information about values reaching their maximum or about changes of properties in other function blocks. To avoid polling, events for automatic notification are defined. Such events must often be sent to several devices (e.g., two HMIs). Because of that, a notification matrix is implemented in every function block. The devices that should be notified of changes to the status of a function are registered in this matrix.

Please note:
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 2-12: Notification matrix (x = notification activated)

The size of a notification matrix depends on the function block, 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, a FktID has 12bits, and that in some function blocks 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.
- Implementation may be done in very economical ways, e.g., by pointers in every function object, that point to DeviceIDs.
- 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.

For very simple function blocks, for example, a CD changer, it is sufficient if the notification matrix provides only three entries for DeviceIDs. A very efficient implementation is possible. 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**. If a controller desires to register, or to remove registration, it sends the following protocol:

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

The DeviceID of the controller is transported at the start of the protocol, as described in section 2.3.5 on page 63, 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 (maximum is 4)
0x2	ClearAll	DeviceID of controller is deleted for all functions
0x3	ClearFunction	DeviceID of controller is deleted for the specified functions (maximum is 4)
Rest	Reserved	

Table 2-13: Parameter Control

On SetFunction and ClearFunction, at most 4 FktIDs can be specified (16 bits each), to avoid exceeding the maximum data length of 12 Bytes of a MOST telegram.

In the table below, the protocols with the different controls for making entries in the notification matrix are listed together with the respective resulting entries.

Protocol	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 2-14: Protocols with different controls for making entries in the notification matrix, and the resulting entries.

Immediately after registration in the notification matrix, the controller receives the status reports of all functions it has activated as events. If a double registering occurs, that is, a device registers that has already been registered, the reports are sent as if the device has been registered for the first time. 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, which activated the respective FktID:

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

Please note:
In case of array properties, only those elements that have been changed are sent as status during notification.

Error handling:

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

- No more registration possible
  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)
  ```

- Device table full
  The device table of the Notification Service is to small. This error may occur when using pointers to DeviceIDs as mentioned above.

  ```
  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 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)
  ```

- 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)
  ```

  This message is also sent, in case a node registers for the property after the problem occurred. Failure of a whole function block is handled in section 3.2.5.5.

Please note:
For keeping the system flexible, and for optimizing the communication effort with respect to the needs, the notifications are re-built at every system start (NetOn).

3 Network Section

3.1 MOST Network Interface Controller and its Internal Services

The MOST Network Interface Controller provides extensive tools for operating the MOST bus simply and safely, and for the transmission of data of different origins. Based on these tools, higher layers are defined. The following sections give an overview of the features of the MOST Network Interface Controller that are available for simplifying the definition of higher layers.

3.1.1 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.

After the MOST Network Interface Controller is reset, the all-bypass is closed. This allows a very fast startup of the system, especially on usage of an optical wakeup mechanism. The all-bypass must be opened by the controlling microcontroller after wakeup of the component.

3.1.2 Source Data Bypass

In order to put source data on the MOST Network, the source data bypass must be opened in the device. That means that the source data is no longer passed through the MOST Network Interface Controller without being processed (that is, not handled by the routing engine RE), but can be routed now, e.g., from a source data port to the bus.

Based on the internal processing of data, a delay of two samples is added in the signal path. The source data bypass should be opened only in devices that put source data onto the bus on runtime.

3.1.3 Master/Slave, Active and Passive Components

Basically, a MOST system consists of up to 64 nodes with identical MOST Network Interface Controllers. By configuration, any of the MOST Network Interface Controllers can be the Timing Master; all the others are slaves. The Timing Master provides generation and transporting of system clock, the frames, and blocks. All Slave devices derive their clock from the MOST bus.

The Timing Master, as well as active Slave devices (source data bypass is open, device can put source data on the bus) add two samples of delay to the path of source data.

A passive Slave device has a closed source data bypass. Since in that case the routing engine is inactive, no delay is generated.

3.1.4 Data Transport

The bit stream is optimized in such a way that processing is easy and maximum functionality is supported. This includes mechanisms for automatic channel routing, network delay detection, and burst data channel management.

The MOST network technology defines an intelligent bit stream, which is capable of providing all MOST network features as described above.

Data is transferred in a continuous bi-phase encoded bit stream yielding more than a 24.8Mbps data rate at a 44.1 kHz rate and a bit error rate of less than 10^{-10}.

Since the MOST system is fully synchronous, with all devices connected to the bus being synchronized to the bus, no memory buffering is needed (unlike isochronous, or asynchronous devices). This keeps cost low.

The sample frequency in a MOST system can be chosen in a range between 30kHz and 50kHz. The frequency depends directly on the application components. Some devices, for example, CD drives, work at a device-specific sample rate. In systems optimized for cost, such devices are regarded as fixed with respect to sample frequency. The sample frequency that is used most should be defined as the system frequency, to avoid sample rate conversion in the different devices.

3.1.4.1 Blocks

Organization of data transfer in blocks of frames is required for network management and control data transport tasks.

A block consists of 16 frames with 512 bits each. Per frame, 60 Bytes of data are available for source data (synchronous and asynchronous packet data), while two Bytes transport control data. The 2 Bytes of 16 frames (1 block) are added to the control frame that transports a control telegram.

3.1.4.2 Frames

The MOST frame structure is designed in a way that provides maximum flexibility in terms of compatibility with a number of existing communication and data transport requirements without any drawbacks in implementation cost or processing overhead. It allows easy re-synchronization, clock and data recovery with the highest data quality and integrity. Built-in structures allow simple network management on the lowest layers avoiding overhead and cost shortcomings.

For synchronization, two different bus node types are required. A Timing Master that generates the frames, and Slave devices that synchronize to the Master clock on the bus.

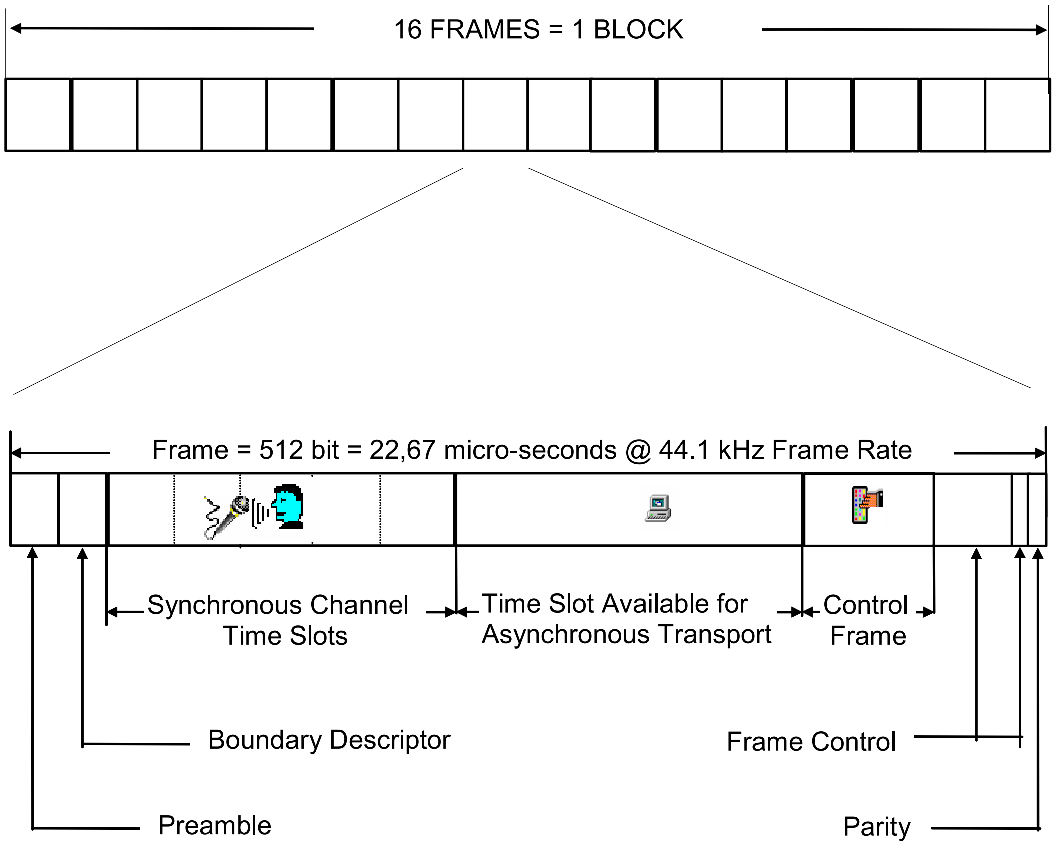

16 FRAMES = 1 BLOCK
Frame = 512 bit = 22,67 micro-seconds @ 44.1 kHz Frame Rate
Synchronous Channel Time Slots
Time Slot Available for Asynchronous Transport
Control Frame
Boundary Descriptor
Frame Control
Preamble
Parity

Figure 3-1: Structure of blocks and frames on the MOST bus

The 64 Bytes (512 bits) wide frame has the following structure:

Byte	Bit	Task
0	0-3	Preamble
0	4-7	Boundary descriptor (synchronous area count value)
1	8-15	Data Byte 0
2	16-23	Data Byte 1
: :	: :	: :
60	480-487	Data Byte 59
61	488-495	Control frame Byte 0
62	496-503	Control frame Byte 1
63	504-510	Frame control and status bits
63	511	Parity bit

Table 3-1: Structure of the MOST frame

3.1.4.2.1 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 Slave and Master nodes. For a Slave 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 Slave node is phase- and frequency-locked to the bit stream, and hence the Master node. In a Master node, the transmitted bit stream is synchronized to an external timing source such as a crystal oscillator, SCK, FSY, or S/PDIF source.

Once all the nodes in the network have locked to the master'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 Slave nodes. The Master 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.

3.1.4.2.2 Boundary Descriptor

The boundary descriptor provides a flexible way of changing the bandwidth for synchronous and asynchronous data transmission. It represents the number of 4 Byte blocks (quadlets) of data used for synchronous data. This value is used to determine the boundary between the synchronous and asynchronous data areas in the frame. A count value of zero indicates no synchronous data and 15 quadlets of asynchronous data, while a count value of 15 indicates 15 quadlets of synchronous data and no asynchronous data.

By this means, a 60 Byte data field can be allocated to either synchronous or asynchronous data on a 4 Byte resolution. As such, it can be optimized to different requirements, depending on the amount of bandwidth required for each type of data.

Note that the maximum number of asynchronous data Bytes per frame is 36 Bytes, which means that the boundary descriptor values can be between 6 and 15.

The boundary descriptor is managed by the Timing Master of a MOST Network. Please note that all synchronous connections must be re-built after having changed the Boundary Descriptor.

3.1.4.2.3 MOST System Control Bits

All other bits within the frame are for management purposes on the network level. While the preamble provides synchronization and clock regeneration, the parity bit indicates reliable data content and is used for error detection and phase lock loop operation.

3.1.4.3 Source Data

3.1.4.3.1 Definition of Control Data and Source Data

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

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

3.1.4.3.2 Differentiating Synchronous and Asynchronous Data

Sixty data Bytes (15 quadlets) total are available for synchronous and asynchronous (packet) data. The number of synchronous and asynchronous Bytes is specified by the boundary descriptor value described above.

3.1.4.3.3 Source Data Interface

The MOST Network Interface Controller can handle a variety of different data formats at its source data port. The source data port formats are controlled via the internal registers of the MOST Network Interface Controller.

3.1.4.3.4 Transparent Channels

In addition to the different transmission procedures, the MOST Network provides a transparent interface (transparent port). This port is over sampled (depending on the system's sample rate, and on the sampling rate chosen for the transparent port) and routed via the network. Therefore source data port 1 is available. It provides, for example, the transparent transmission of a RS232 interface, i.e., without synchronizing RS232 to the bus.

If no transparent channel is required, source data port 1 can be used as a standard source data port.

3.1.4.3.5 Synchronous Area

The synchronous channel time slots are available for real-time data such as audio/video or sensors 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 channels for a certain period of time (e.g., while playing an audio source). The bandwidth for such a channel can be adjusted by allocating any number of Bytes to one logical channel. The maximum number of Bytes available in a synchronous channel is 60 Bytes/frame, which is corresponding to 60 x 8 bits or 15 stereo channels of CD-quality audio. The typical frame rate is 44,100 frames/second.

The routing engine (RE) is used to route data to and from the appropriate sources or sinks within a node. Internal synchronization is provided so input data does not need to be phase-aligned to the MOST Network Interface Controller. The RE provides full flexibility in directing data from any source to any sink just by setting the appropriate value in the corresponding registers.

3.1.4.3.6 Asynchronous (Packet Data) Area

Another time slot is available for asynchronous data transport as required for more packet-oriented, burst-like data. In contrast to the control data channel, the asynchronous data channel provides transmission of longer data packets.

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 an asynchronous channel when using the 48 Bytes data link layer, is 48 Bytes. In case of using an alternative data link layer, the maximum packet length is 1014. The data on this channel is CRC protected. The asynchronous message is defined as follows:

Byte	Task
0	Arbitration
1-2	Target address
3	Length (in Quadlets = 4 Bytes)
4-5	Own address (Source address)
6-53	Data area
54-57	CRC

Table 3-2: Structure of a frame in the asynchronous area (48 Bytes data link layer)

Byte	Task
0	Arbitration
1-2	Target address
3	Length (in Quadlets = 4 Bytes)
4-5	Own address (Source address)
6-1019	Data area
1020-1023	CRC

Table 3-3: Structure of a frame in the asynchronous area (alternative data link layer)

Since the asynchronous 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 CRC is provided. The CRC is calculated in the background and can be indicated in a register at the end of each asynchronous message. A low-level retry mechanism is not implemented.

3.1.4.4 Control Data

3.1.4.4.1 Control Data Interface

The transmission of control data to and from the MOST Network Interface Controller is done via the control bus.

3.1.4.4.2 Description

The control data is used mainly for communication between the single nodes of the bus. This is where commands, status and diagnosis messages, as well as gateway messages are handled. The protocol on this channel runs in a carrier sense multiple access (CSMA) manner offering predictable response times, which are considered essential in an audio/video control network. At a system sample frequency of 44.1 kHz, 2756 messages per second are transmitted, which corresponds to a gross data rate of 705.6 kBit/s.

Since 2 out of 64 messages are used for a system wide distributing of the allocation information by the network, the number of messages per second available for control messaging is 2670. When subtracting the data used for control and data securing, the net data rate (user data plus addressing) is 405.84 kBit/s, which corresponds to 19 Bytes per message that can be read from the MOST Network Interface Controller.

A MOST device can access every third message propagated through the network. So a single device has a maximum message rate of 890 per second, or a net data rate of 135.28kBit/s

There are two kinds of control messages. Normal messages provide control of applications, while system messages handle system-related operations such as resource handling. A control data message is 32 Bytes long and has the following structure:

Byte	Task
0-3	Arbitration
4-5	Target address
6-7	Own address (Source address)
8	Message type
9-25	Data area
26-27	CRC
28-29	Transmission status
30-31	Reserved

Table 3-4: Structure of a control data frame

Please note:
The delay time between two messages in case of low level retries – must be identical in all nodes of a MOST Network.

Message type:
Normal messages:
- Single cast (logical or physical addressing)
- Groupcast
- Broadcast

System messages:
- Resource Allocate
- Resource De-Allocate
- Remote GetSource

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 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 controlling device (e.g., external micro controller).

3.1.5 Internal Services

3.1.5.1 Addressing

The MOST Network Interface Controller supports four different ways of addressing:

- Node position in the ring.
  The node position is generated automatically in each node during the locking procedure of the MOST Network.
- Unique node address (2 Bytes).
  This address can be set by the application.
- Group address (1 Byte).
  Group address can be set by the application. A group is made up of devices that have the same number in the group address register.
- Broadcast
  The broadcast address is a special group address. When used, the message 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.

The different ways of addressing are mapped into the address area of a MOST Network Interface Controller:

Address range	Mode
0x0000…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…0x04FF	Node position (0x0400 + POS) address range.
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
0xFFFF	Uninitialized logical node address
	Note: the highest nibble is reserved for future use

Table 3-5: Addressing modes vs. address range

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 definition of the system.

3.1.5.2 Support at System Startup

The MOST Network Interface Controller meets all requirements of a low level startup. Several supporting mechanisms are provided. All components of the system get a unique number, with numbering starting at the Timing Master at 0x00, and then incremented by one. These numbers can be 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 wakeup mechanism.

3.1.5.3 Delay Recognition

Based on the fact that every node may be active or passive with respect to source data handling (source data bypass open or closed), and that every active node generates two samples of delay, it is useful to have information about source data delay, for example, for noise compensation applications, or in high-end audio applications.

Therefore a mechanism is implemented in each MOST Network Interface Controller, giving access to information about the total delay of the system, and to the delay up to the local node with respect to the Timing Master.

3.1.5.4 Automatic Channel Allocation

Since administration of up to 30 audio channels would need many resources on an application's side, the MOST system supports resource administration on the MOST Network Interface Controller level.

Allocating one or more audio channels (up to 64 bits per allocation procedure) is done via a request from an application to the Timing Master of the network. If there are enough channels, the application will get a handle, by which source data can be routed onto the network. The handle can also be used for de-allocating. A channel resource allocation table, distributed automatically in the ring (on the MOST Network Interface Controller level), gives access to the current allocation status of the channels in each node.

The channel map that belongs to the handle can be retrieved from the MOST Network Interface Controller, or it can be delivered during connection management via control messages. It is possible to change allocation during runtime.

3.1.5.5 Detection of Unused Channels

Detection of unused channels, i.e., channels, that are allocated by a device, but which are no longer used, is done with the help of the channel resource allocation table. Only the timing master can determine from its channel resource allocation table if there are unused channels. If there are unused but allocated channels, they should be de-allocated with respect to the network resources.

3.2 Dynamic Behavior of a Device

3.2.1 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 (or the dynamic of the network interface of a device). The expression NetInterface stands for the entire communication section of a device, that is, the optical interface, the MOST Network Interface Controller, and the Network Service.

The figure below shows a layer model of a device. The lowest layer is the power supply. On this layer, every hardware function is built, that is, the hardware of the NetInterface, which is made up of the MOST Network Interface Controller, the optical interface, and the controller on which the Network Service are running. The Network Service make up the next layer, on which the higher services of address management, power management and network error management are based. At the top layer there is the application itself.

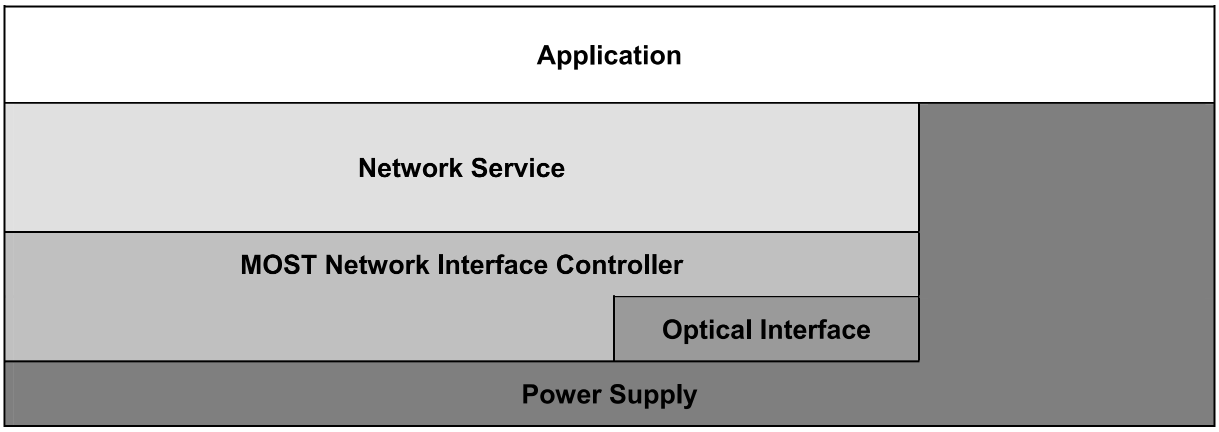


Figure 3-2: Layer model of a device

Generally, for each device, the device specification must define all the possible combinations of the states of the application section and the communication section. Especially from the view of the network, there are three states that are mandatory for each device:

1. **DevicePowerOff:** Communication section is in state NetInterfacePowerOff. The application section in a non-waking device is in state ApplicationPowerOff, or in a waking device in state ApplicationSleep.
2. **DeviceStandBy:** This state is mainly influenced by state ApplicationLogicOnly. 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 NetInterfaceNormalOperation.
3. **DeviceNormalOperation:** The communication section as well as the application section are in state NormalOperation.

Since these main states are only a few of all possible device states, it is not useful to use a diagram. 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, e.g., by a timer, can check an external signal, and return to state ApplicationSleep without waking up the NetInterface. The device does not leave the mode.
- The application may be awakened, e.g., 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 by light 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 light on the bus is switched off, the device changes to state DevicePowerOff.

DeviceNormalOperation:

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

The following description of the dynamic behavior is done from the bottom up. The most significant subjects regarding power supply are described in section 4.1 on page 203. The following section focuses on the dynamic behavior of the NetInterface.

3.2.2 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 following figure shows the states of the NetInterface and the events that lead to state transitions. The following sections explain the individual states.

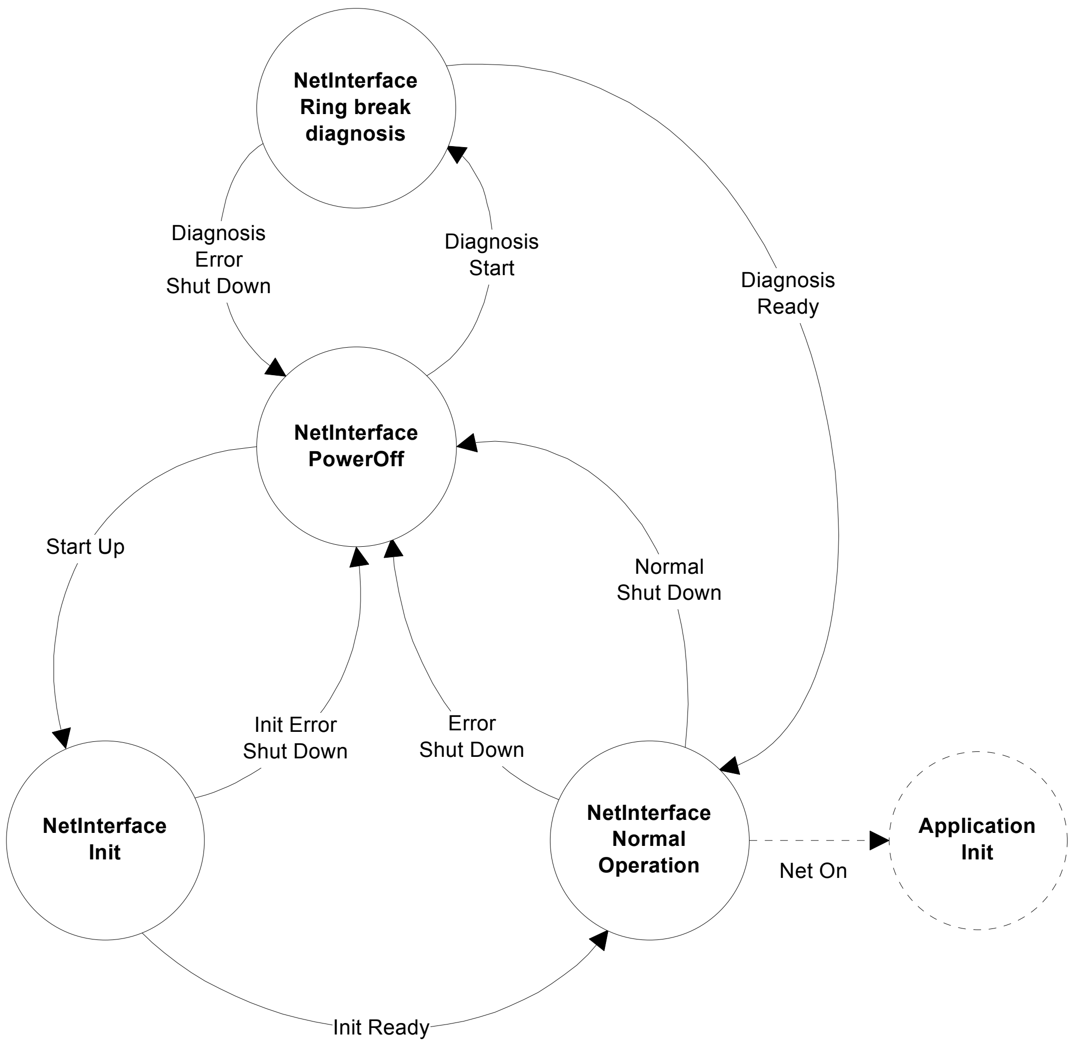


Figure 3-3: Flow chart "Overview of the states in NetInterface"

3.2.2.1 NetInterfacePowerOff

In state NetInterfacePowerOff, the NetInterface is switched off from the view of the network. The FOT does not emit light. 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., RMCK generation).

State NetInterfacePowerOff is left when one of the following events occurs:

Event	Transition to	Cause
Start Up	NetInterfaceInit	A NetInterface is activated either by light at the receiving FOT, by the application (Hypothetical example: phone receives a call), or by a switch at the device.
Diagnosis Start	NetInterfaceRingBreakDiagnosis	A NetInterface is activated by connecting to power (for information about signal SwitchToPower please refer to section 4.1 on page 203)

Table 3-6: Events in state NetInterfacePowerOff

3.2.2.2 NetInterfaceInit

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

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

Event	Transition to	Cause
Init Ready	NetInterfaceNormalOperation	NetInterface is ready for communication (see below).
Init Error Shut Down	NetInterfacePowerOff	Error occurred during initialization (see below).

Table 3-7: Events in state NetInterfaceInit

Causes for event Init Ready:

- In the Master device:
  Net Activity and stable lock (at minimum for time t_{Lock}) were recognized. Lock is called stable if for a period of time t_{Lock} no unlock events occurred.
- In a Slave device:
  Stable lock (at minimum for time t_{Lock}) was recognized and the Boundary Descriptor value has a valid value (>5). This fact is the basis for the statement that the Timing Master of the system has also recognized stable lock, and the ring is closed.

Causes for event Init Error Shut Down:

- In the Master device:
  Timeout t_{Config}, occurs before a stable lock can be recognized.
- In a waking Slave device:
  Timeout t_{Config}, occurs before a closed ring can be recognized. Error Error_NSInit_Timeout is stored by the application.
- In a non-waking Slave device:
  Timeout t_{Config}, expires before light was recognized, or a closed ring was recognized; or the light was switched off again.

In a Slave device (non-waking) the all-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 Slave device and the Master device, the all-bypass is deactivated immediately after having entered this state (light 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 NetInterfaceInit. A differentiation is made between Master and Slave. On this level, Master means Timing Master and Slave means Timing Slave.

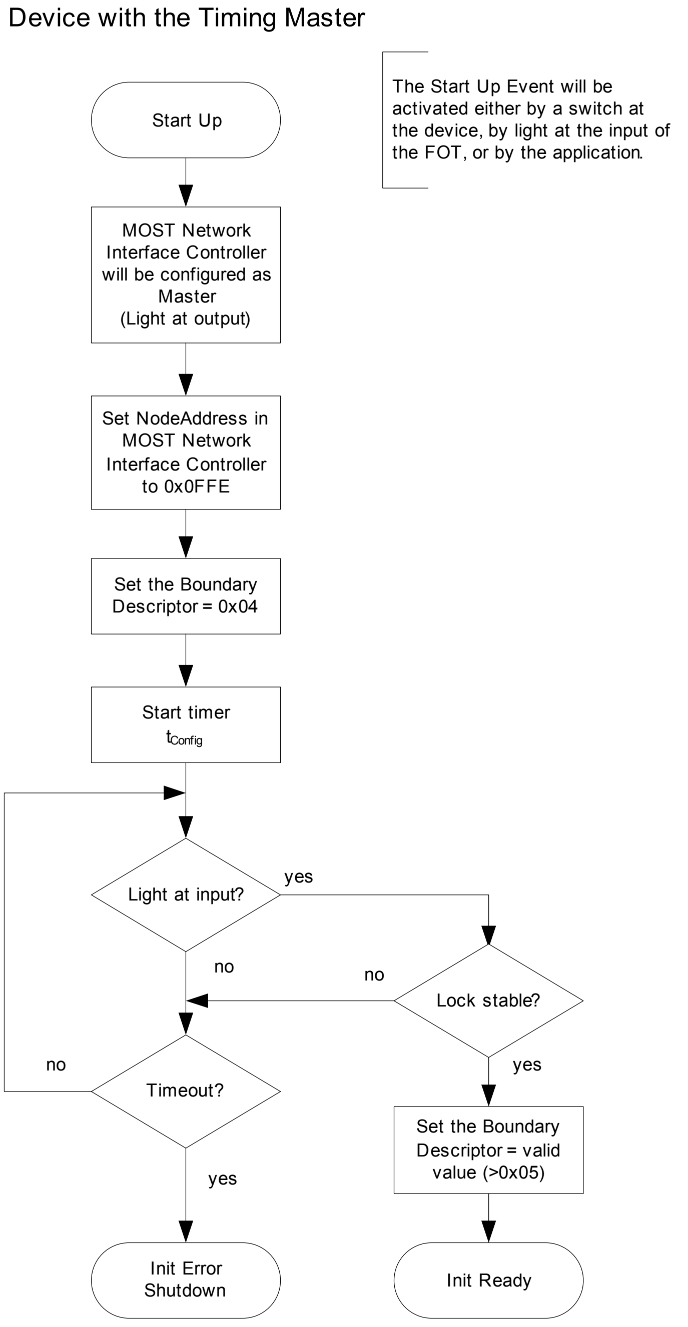

Device with the Timing Master
The Start Up Event will be activated either by a switch at the device, by light at the input of the FOT, or by the application.
Start Up
MOST Network Interface Controller will be configured as Master (Light at output)
Set NodeAddress in MOST Network Interface Controller to 0x0FFE
Set the Boundary Descriptor = 0x04
Start timer t_Config
Light at input?
yes
no
Lock stable?
no
yes
Timeout?
no
yes
Set the Boundary Descriptor = valid value (>0x05)
Init Error Shutdown
Init Ready

Figure 3-4: Behavior of a Master device in state NetInterfaceInit

When entering state NetInterfaceInit, the Timing Master loads the Boundary Descriptor with the "invalid" value 0x04. This value is transferred to all MOST Network Interface Controllers via the frame. As soon as the Timing Master recognizes a stable lock, it sets the Boundary Descriptor to a valid value (>0x05). By doing this, every Slave in the ring can recognize when the Timing Master has reached stable lock.

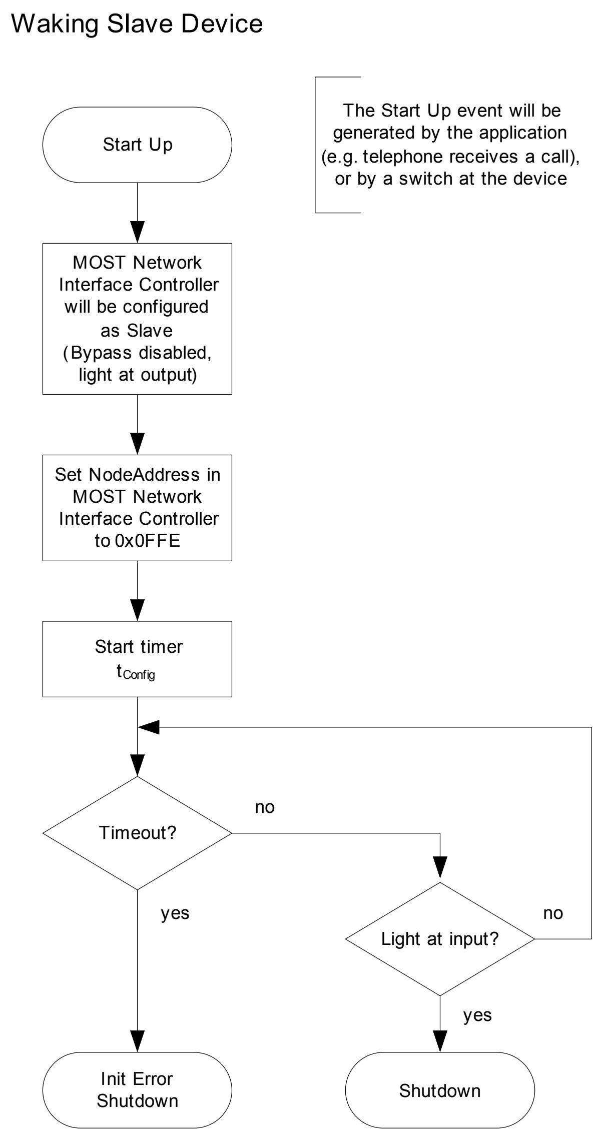

Waking Slave Device
Start Up
The Start Up event will be generated by the application (e.g. telephone receives a call), or by a switch at the device
MOST Network Interface Controller will be configured as Slave (Bypass disabled, light at output)
Set NodeAddress in MOST Network Interface Controller to 0x0FFE
Start timer t_{Config}
Timeout?
no
yes
Light at input?
no
yes
Init Error Shutdown
Shutdown

Figure 3-5: Behavior of a waking Slave device in state NetInterfaceInit

After having woken the ring (the light returned from Timing Master), the Slave Device goes to Shutdown. From there it starts up as a standard Slave Device, woken by the Timing Master.

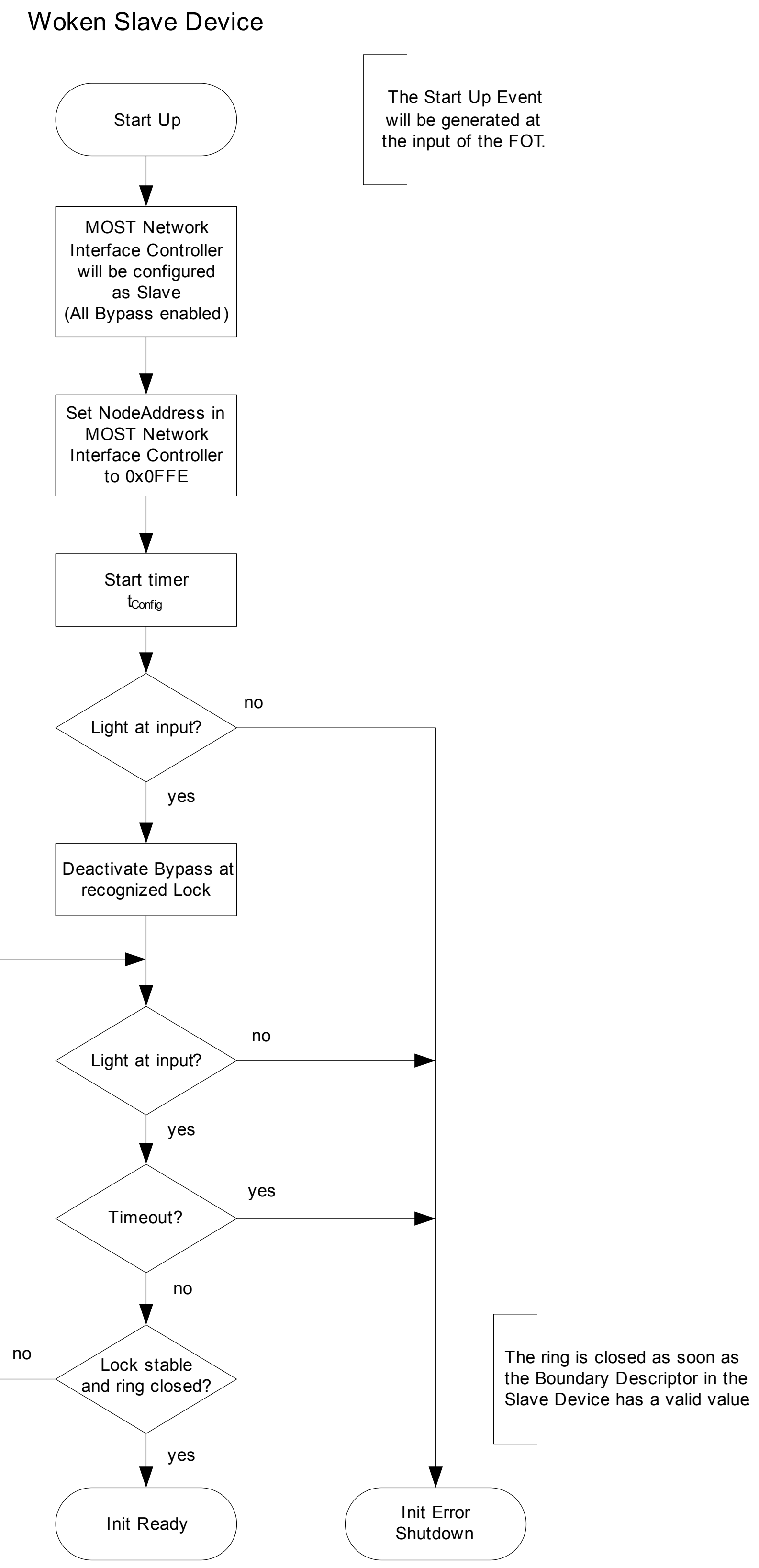

Woken Slave Device
Start Up
The Start Up Event will be generated at the input of the FOT.
MOST Network Interface Controller will be configured as Slave (All Bypass enabled)
Set NodeAddress in MOST Network Interface Controller to 0x0FFE
Start timer t_{Config}
Light at input?
no
yes
Deactivate Bypass at recognized Lock
Light at input?
no
yes
Timeout?
yes
no
Lock stable and ring closed?
no
yes
The ring is closed as soon as the Boundary Descriptor in the Slave Device has a valid value
Init Ready
Init Error Shutdown

Figure 3-6: Behavior of a woken Slave device in state NetInterfaceInit

3.2.2.3 NetInterfaceNormalOperation

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.

Examples for initializing a higher layer due to NetOn Event:

- Check of system configuration and building of the Central Registry (refer to section 3.3.3).
- Setting of the logical node address and group address (refer to section 3.3).
- 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 Shut Down	NetInterfacePowerOff	NetInterface will be deactivated by switching off light.
Error Shut Down	NetInterfacePowerOff	NetInterface will be deactivated due to a critical unlock.
Net On	Report to an application	Entering state NetInterface Normal Operation

Table 3-8: Events in state NetInterfaceNormalOperation

The Normal Shut Down event is generated as soon as no light is recognized at the input.

In state NetInterfaceNormalOperation, the Network Service checks the lock state of the PLL of the MOST Network Interface Controller. On an unlock, the application is informed as soon as possible by an unlock event. Every application then has to save its output signals (e.g., amplifier mutes its outputs).

In addition to that, 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 Shut Down 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 cause a critical unlock) an Error Shut down 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 3.9 on page 196).

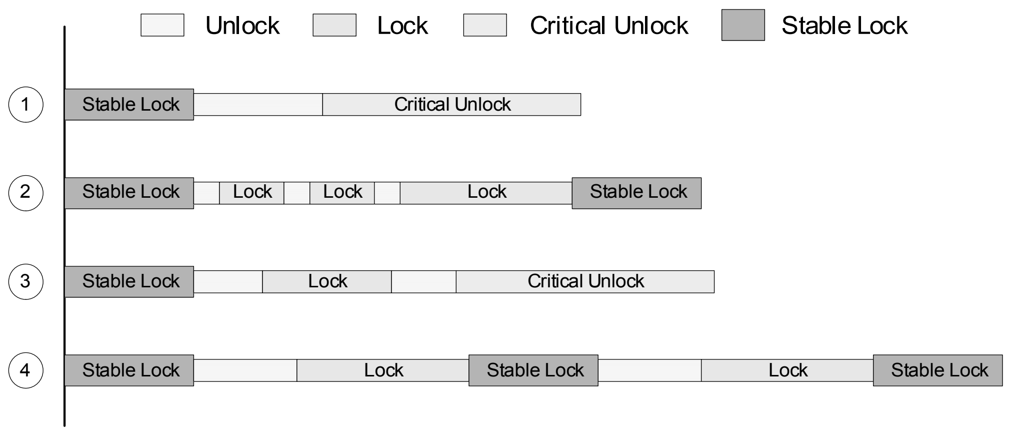


Figure 3-7: Examples of the behavior when unlocks occur.

1. The first example shows an unlock that persists longer than t_{Unlock}. This results in a critical unlock (Error Shutdown event).
2. The second example shows a series of short unlocks with an accumulated total that is less than t_{Unlock}. In this case no critical unlock will occur.
3. The third example shows when two unlocks with an accumulated total that exceeds t_{Unlock}. This leads to a critical unlock.
4. The unlocks in the last example 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 between them.

In addition to that, the change in number of MOST devices is checked. If this number indicates a Network Change Event, the application will get informed about that.

The flow chart below shows the behavior in state NetInterfaceNormalOperation:

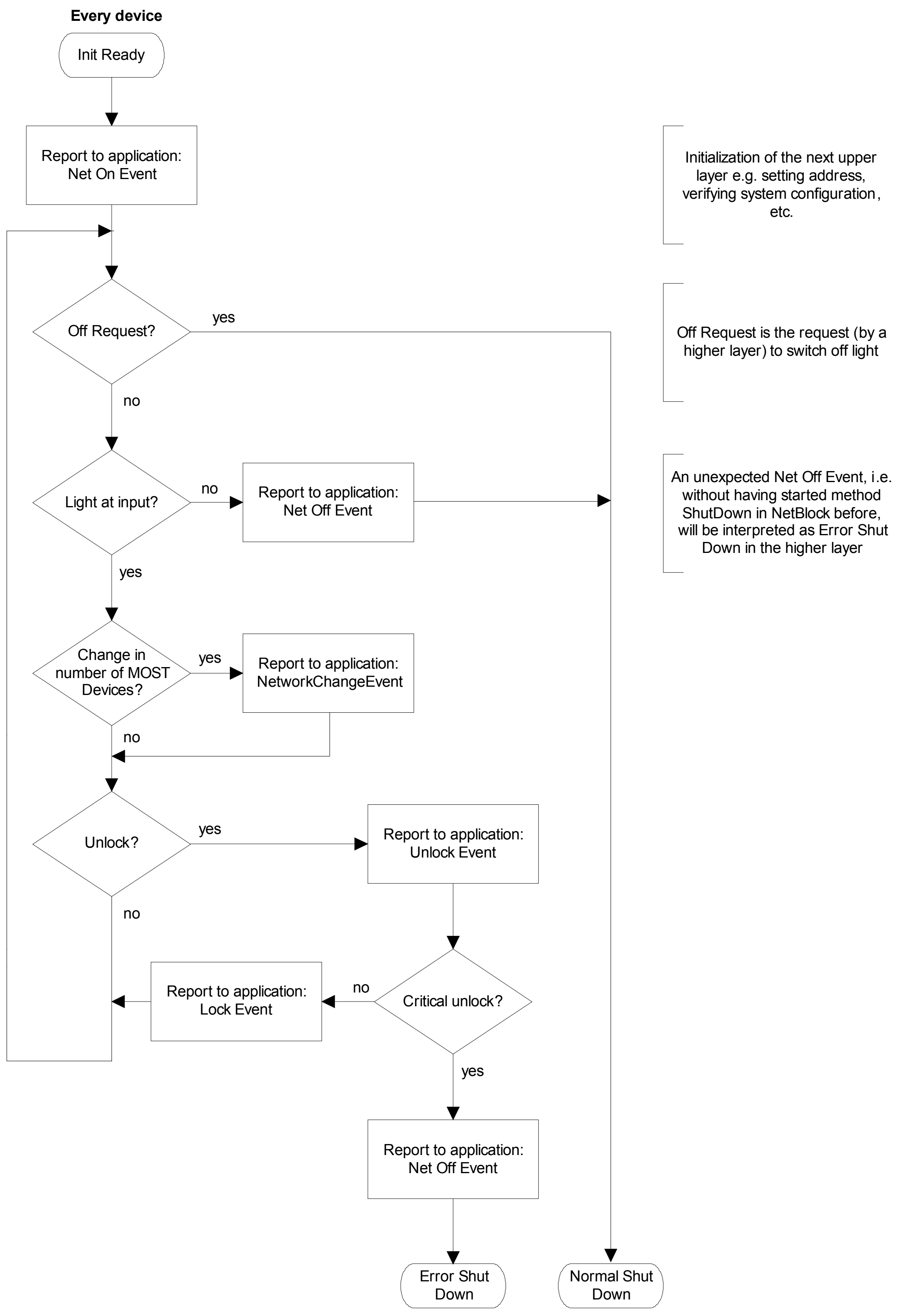

Every device
Init Ready
Report to application: Net On Event
Initialization of the next upper layer e.g. setting address, verifying system configuration, etc.
Off Request?
yes
no
Off Request is the request (by a higher layer) to switch off light
Light at input?
no
Report to application: Net Off Event
yes
An unexpected Net Off Event, i.e. without having started method ShutDown in NetBlock before, will be interpreted as Error Shut Down in the higher layer
Change in number of MOST Devices?
yes
Report to application: NetworkChangeEvent
no
Unlock?
yes
Report to application: Unlock Event
no
Report to application: Lock Event
no
Critical unlock?
yes
Report to application: Net Off Event
Error Shut Down
Normal Shut Down

Figure 3-8: Behavior in state NetInterfaceNormalOperation

3.2.2.4 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. It is run not during normal operation, but in the car repair, or at the assembly line.

The RingBreakDiagnosis process can be started by various triggers, which must be chosen and implemented by the System Integrator. One possible way is, to start the RingBreakDiagnosis by disconnecting the System from the power source for a short time. In this case, RingBreakDiagnosis 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). This signal is not evaluated during NetOn. If SwitchToPower is used to trigger RingBreakDiagnosis, all devices must start the diagnosis within $t_{Diag\_Start\_}$.

In state NetInterfaceRingBreakDiagnosis the network cannot reach normal operation. In this state, 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 immediately changes to state NetInterfaceNormalOperation.

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 Timing Master at the end of RingBreakDiagnosis (since there was no light at its input).

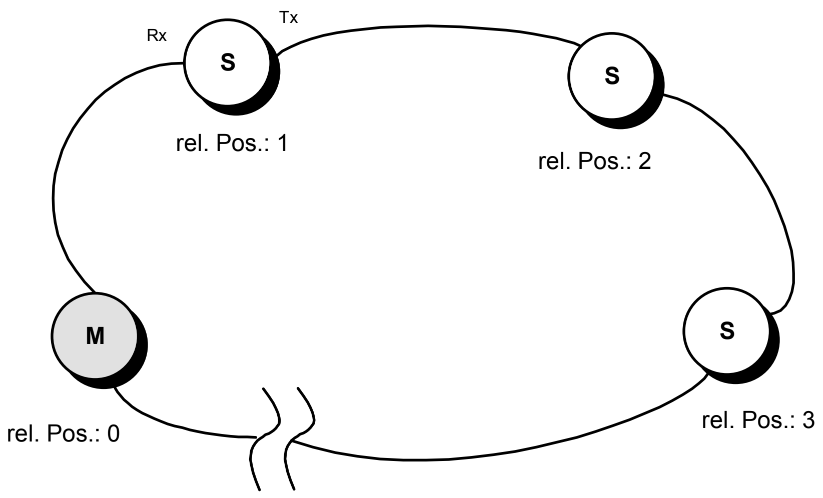


Figure 3-9: Localizing a fatal error with the help of ring break diagnosis.

Event	Transition to	Cause
Diagnosis Ready	NetInterfaceNormalOperation	No fatal error.
Diagnosis Error Shut Down	NetInterfacePowerOff	Fatal error (Ring break or defective device)

Table 3-9: Events in state NetInterfaceRingBreakDiagnosis

During RingBreakDiagnosis a device stays configured as Timing Master, until it recognizes light at its input, or until the Diagnosis Error Shut Down event is generated by occurrence of the timeout (t_{Diag_Master} or t_{Diag_Slave} respectively). On a fatal error, the application stores the error Error_Ring_Diagnosis with the relative ring position.

After recognition of a stable lock, a Timing Master device generates a Diagnosis Ready event and changes immediately to state NetInterfaceNormalOperation.

The timeout values (t_{Diag_Master} or t_{Diag_Slave}) can be changed through the system integrator, if alternative approaches for ring break diagnosis are used. In this case, the system integrator must make sure that all devices in the network are able to start the diagnosis process within the specified timeouts.

As soon as a device, which does not contain the Timing Master under normal operation conditions, recognizes light at its input, it is configured as Slave (all-bypass enabled). The all-bypass is deactivated after a recognized lock. If no lock errors occur for a time t_{Diag_Lock} (stable lock), the relative ring position is determined.

If, on stable lock, the Boundary Descriptor value is greater than 5, the ring is closed. There is no defect and the NetInterface changes to state NetInterfaceNormalOperation.

If the ring could be closed, every NetInterface switches to state NetInterfaceNormalOperation. The application will get notified about that by the NetOnEvent. After that, all high level initializations must be performed (Building of the Central Registry, address initialization, notification...).

If the ring could not be closed since a ring break exists, devices should not be restarted by incoming light until $t_{Diag\_Restart}$ has passed.

The following flow charts show the behavior in the state NetInterfaceRingBreakDiagnosis:

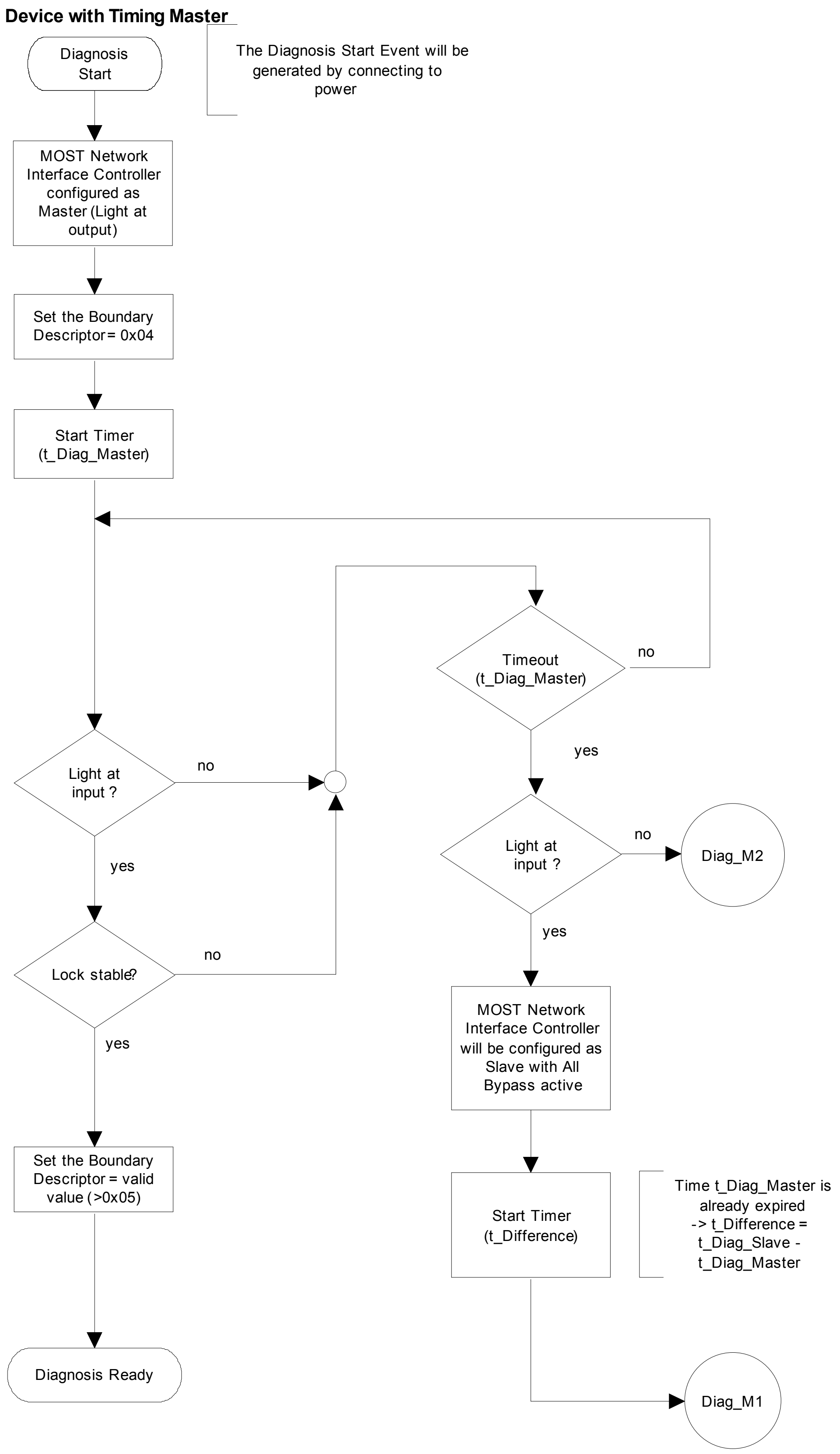

Device with Timing Master
Diagnosis Start
The Diagnosis Start Event will be generated by connecting to power
MOST Network Interface Controller configured as Master (Light at output)
Set the Boundary Descriptor = 0x04
Start Timer (t_Diag_Master)
Timeout (t_Diag_Master)
no
yes
Light at input ?
no
yes
Lock stable?
no
yes
Light at input ?
no
Diag_M2
yes
MOST Network Interface Controller will be configured as Slave with All Bypass active
Start Timer (t_Difference)
Time t_Diag_Master is already expired -> t_Difference = t_Diag_Slave - t_Diag_Master
Set the Boundary Descriptor = valid value (>0x05)
Diagnosis Ready
Diag_M1

Figure 3-10: Behavior during ring break diagnosis in a Timing Master (part 1)

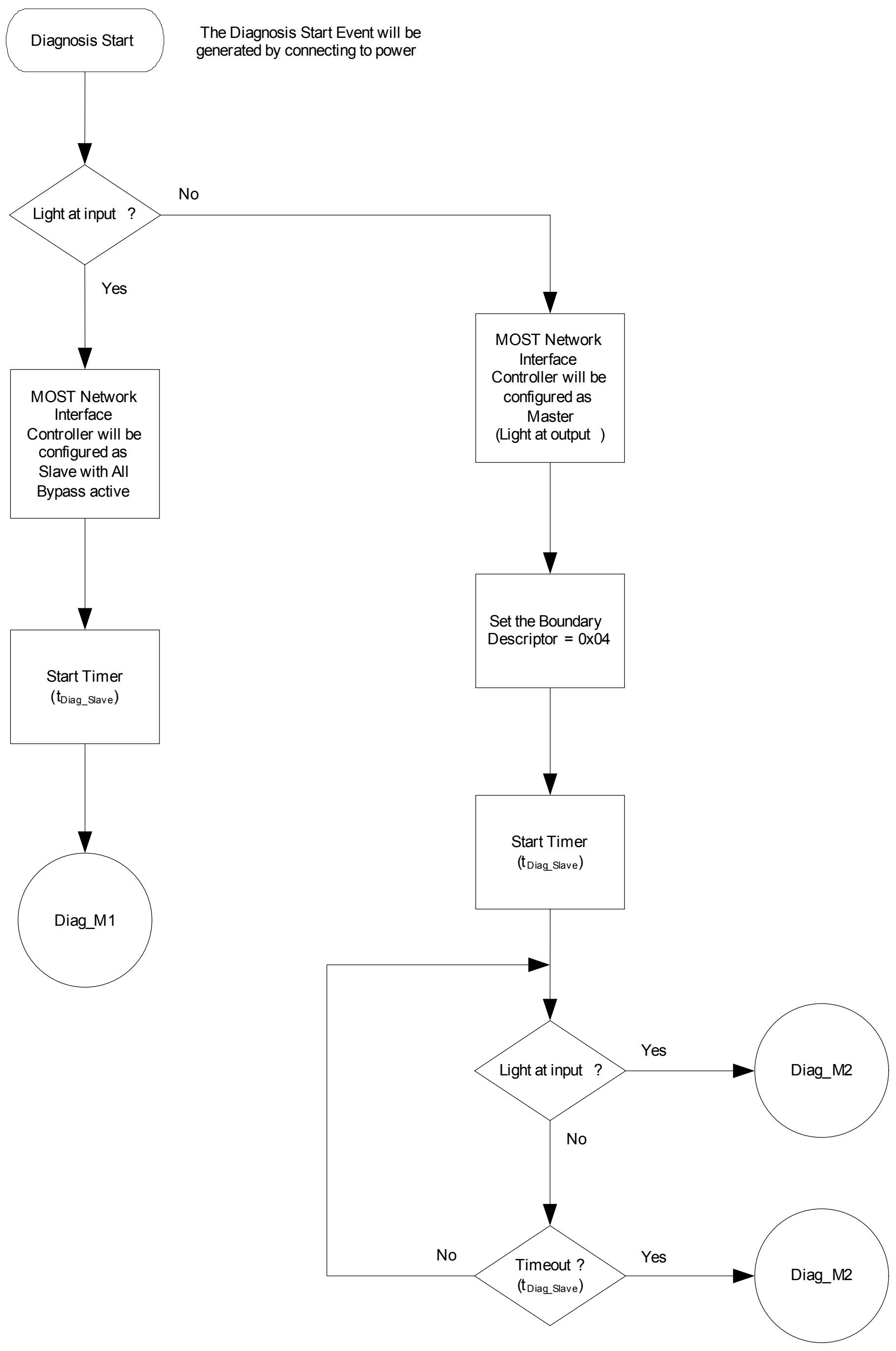

Diagnosis Start
The Diagnosis Start Event will be generated by connecting to power
Light at input ?
No
Yes
MOST Network Interface Controller will be configured as Slave with All Bypass active
MOST Network Interface Controller will be configured as Master (Light at output)
Set the Boundary Descriptor = 0x04
Start Timer (t_{Diag_Slave})
Start Timer (t_{Diag_Slave})
Diag_M1
Light at input ?
Yes
Diag_M2
No
No
Timeout ? (t_{Diag_Slave})
Yes
Diag_M2

Figure 3-11: Behavior during ring break diagnosis in a Slave (part 1)

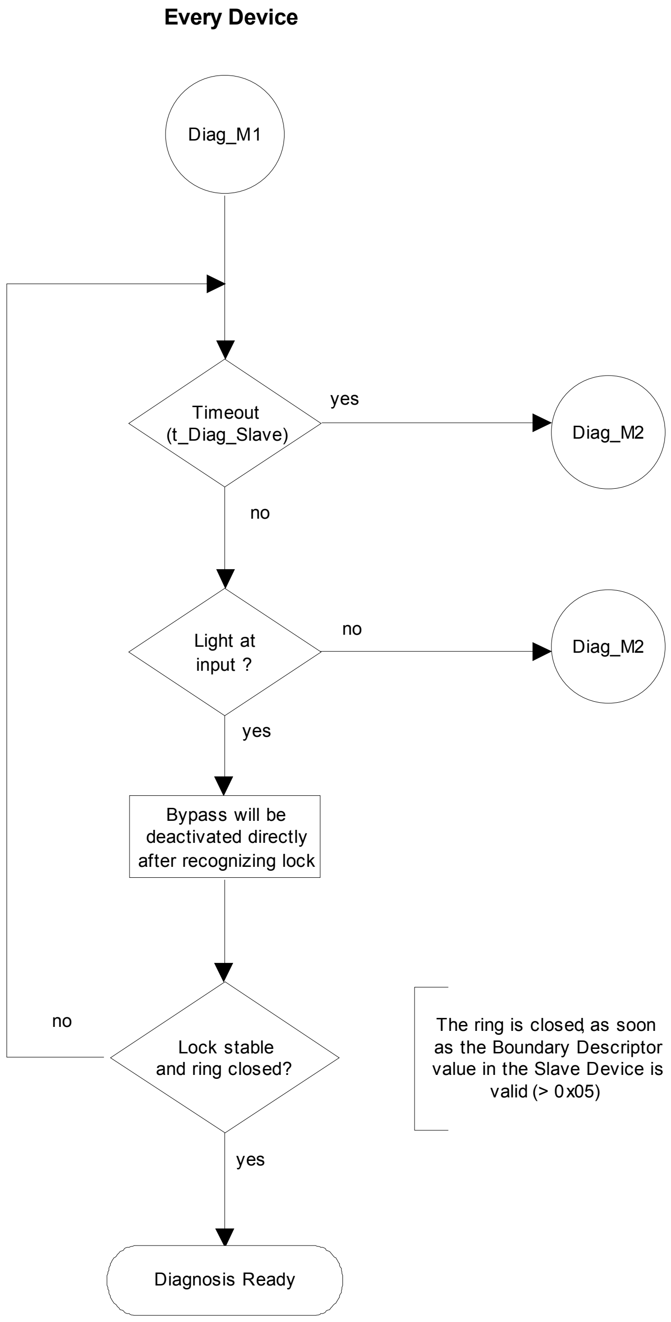


Figure 3-12: Behavior during ring break diagnosis in a Timing Master and Slave (part 2)

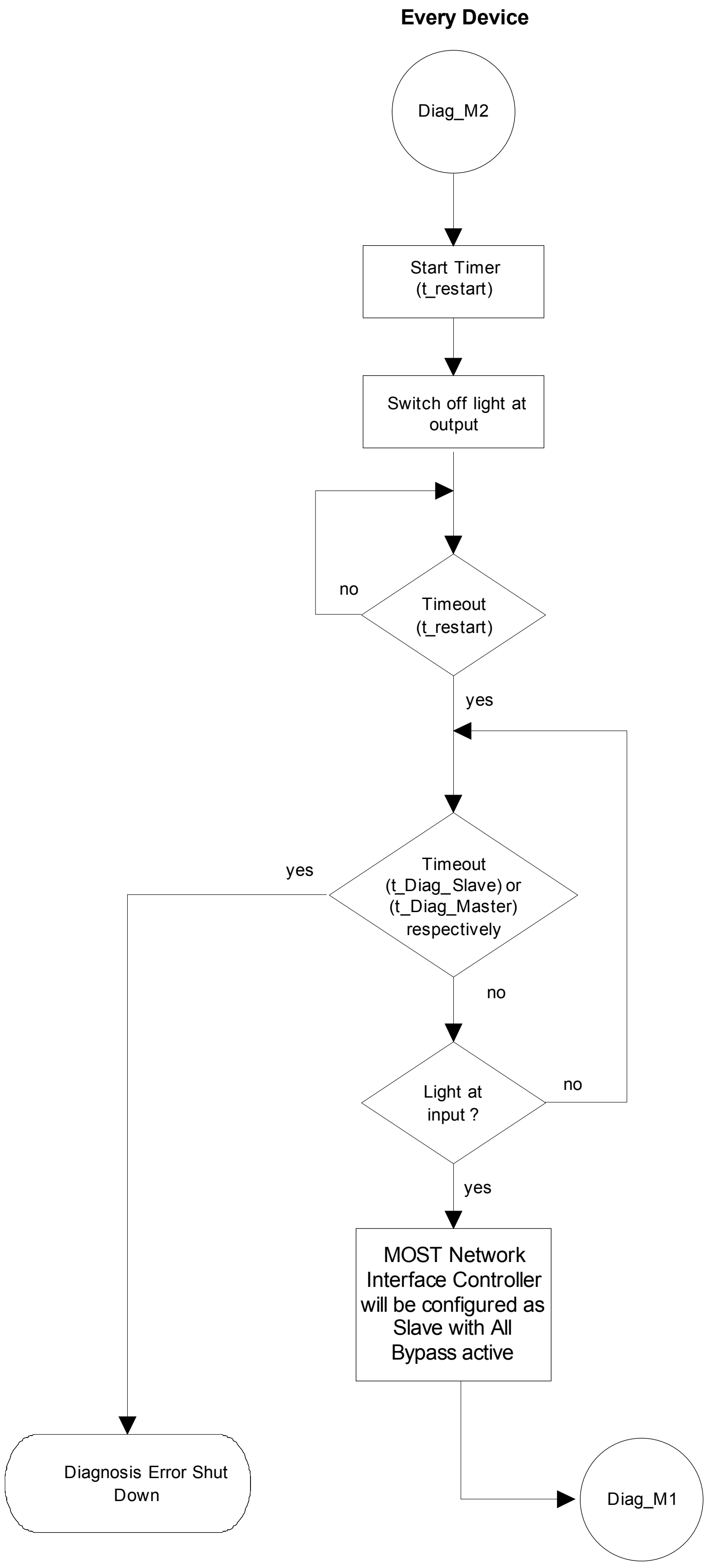

Every Device
Diag_M2
Start Timer (t_restart)
Switch off light at output
no
Timeout (t_restart)
yes
yes
Timeout (t_Diag_Slave) or (t_Diag_Master) respectively
no
Light at input ?
no
yes
MOST Network Interface Controller will be configured as Slave with All Bypass active
Diagnosis Error Shut Down
Diag_M1

Figure 3-13: Behavior during ring break diagnosis in a Timing Master and Slave (part 3)

3.2.3 Secondary Nodes

For some applications it can be useful to integrate two MOST Network Interface Controllers into one device. The two nodes are called "Primary" and "Secondary" node. A detailed definition of these names and a description of the possible structures are available in section 3.10 on page 200. A Secondary Node does not contain any function blocks. In case of receiving any request, it returns an error (Error Secondary Node. Please refer to section 2.3.2.5.1 on page 45). This error has the meaning of "I am a Secondary Node only. The node responsible for me has DeviceID 0xnnnn". The example below shows this mechanism by means of the request of System Configuration (NetworkMaster). During Network Configuration request, the NetworkMaster asks all nodes for the function blocks they contain. The secondary node therefore replies:

```
SN -> NM: NetBlock.Pos.FBlockIDs.Error (ErrorCode = 0x0A = Secondary Node,
                                        ErrorInfo = DeviceID of Primary Node)
```

In the Central Registry it must be marked, which node is the Primary Node to a certain Secondary Node. Therefore, the Central Registry must be sorted in a way, that the Secondary Node's entry directly succeeds the entry of the Primary Node in case of a request. This does explicitly not refer to the hardware configuration in the respective device. In a MOST device, the Primary Node can be arranged behind the Secondary Node as well.

For completing the entries of Secondary Nodes in the Central Registry, each Secondary Node is registered with a single Function Block (FBlock) having FBlockID.InstID = 0xFC.0.

3.2.4 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.

3.2.4.1 Waking of the Network

Waking the network is done by emitting modulated light (light on). In principle the network can be awakened by any node. The ability of a node to wake up the network can be activated or deactivated by the PowerMaster (e.g., in case of a critical charge status of the accumulator) in the property **AbilityToWake**, which is mandatory for every NetBlock. 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).

Please note:
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 an electrical wakeup line. It is up to the system integrator to choose the preferred wakeup method. The process described here, is independent of the wakeup method.

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

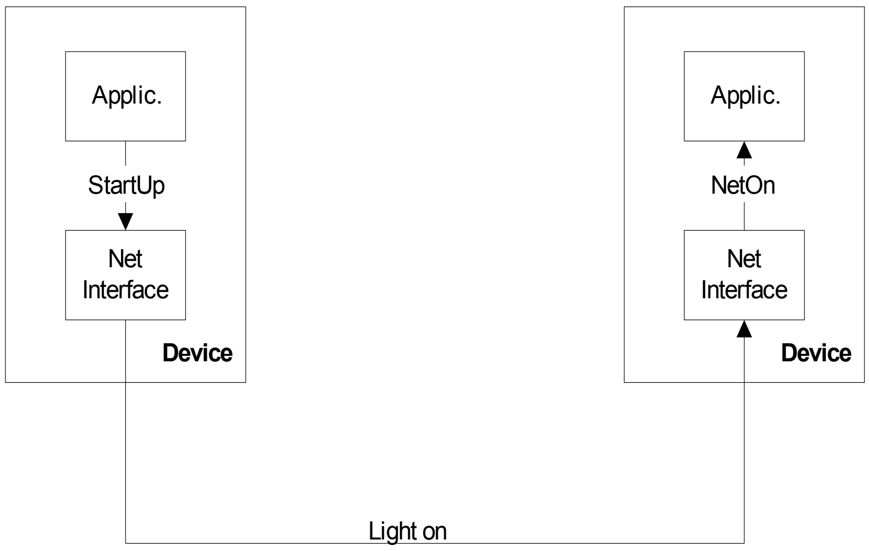


Figure 3-14: Example (2 devices) for waking of the MOST network via light on the network

3.2.4.2 Network Shutdown

Switching off the network is done on lowest level by switching off light. A device, which has switched off light, must not switch it on again before $t_{Restart}$ occurred. This applies even if it recognizes light at its input (optical wakeup). Electrical wakeups may be latched to perform a wakeup after $t_{Restart}$.

All devices, except the one containing the PowerMaster, switch off light when certain errors occur (unlock, low voltage with reset). This is done without warning the other devices by sending a telegram.

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 $t_{Suspend}$ before it actually shuts down 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 light to be switched off.

The PowerMaster switches off light $t_{ShutDownWait}$ after ShutDown.Start(Execute). This time allows to shutdown audio output without audible side effects. If the light was not switched off within $t_{SlaveShutdown}$ a slave device may switch off the light.

If a function block desires to communicate, it must notify the PowerMaster after ShutDown.Start(Query) with ShutDown.Result (Suspend) within time $t_{Suspend}$. The PowerMaster then postpones its attempt to switch off for time $t_{RetryShutDown}$, before retrying to shut down. This procedure guarantees that a device, which woke the bus in the parked vehicle, does not need to prevent the PowerMaster from switching off the network actively (according to the current status of the vehicle). For switching off, the PowerMaster calls the respective routine in the Network Service. The status “light off” travels around the ring in the same way as “light on” when waking the network. After a certain delay time $t_{PwrSwitchOffDelay}$ the nodes change to sleep mode.

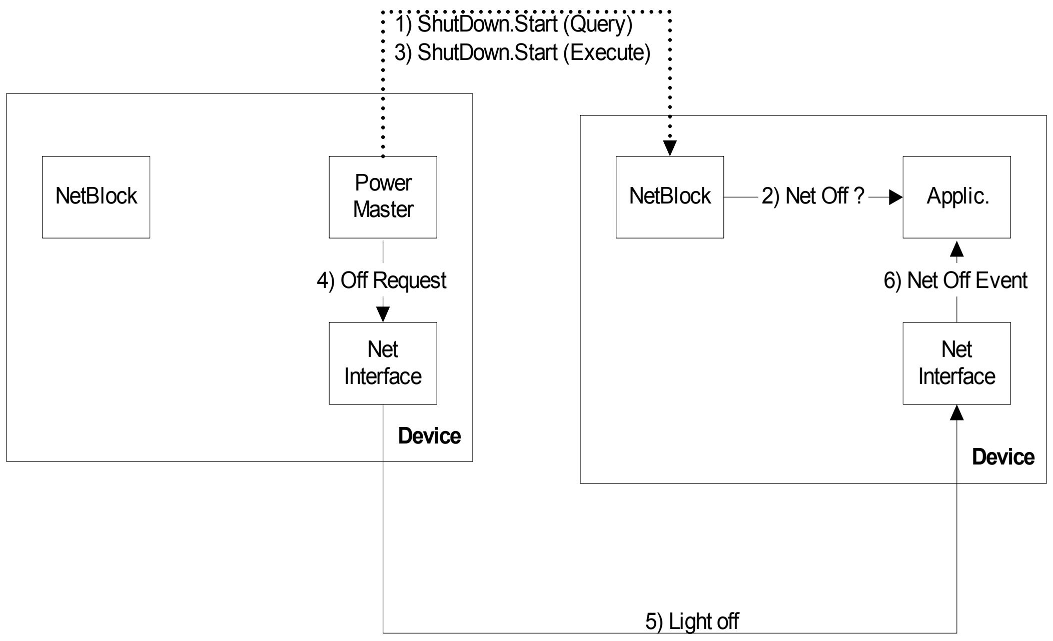


Figure 3-15: Switching off MOST Network via starting method ShutDown in every NetBlock, and signaling to every application, and switching off light

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

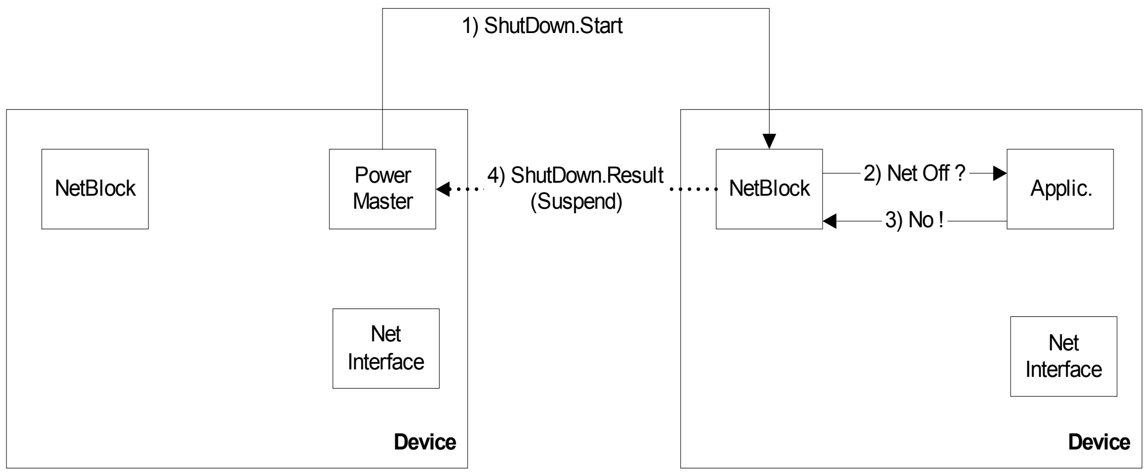


Figure 3-16: Prevention of switching off MOST Network via ShutDown.Result (Suspend)

Please note:
If the light is switched off during shutdown, e.g., by low voltage, long unlock or fatal error, the PowerMaster must not wake the network for being able to finish its shutdown procedure. The PowerMaster must regard the shutdown procedure as complete.

3.2.4.3 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 affects 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 full functionality. Also, the Device has to know the current System State when it wakes from Device Shutdown; therefore the Network Master 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 3.2.4.2 for information about Network Shutdown.

3.2.4.3.1 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 function blocks are communicating with function blocks 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 $t_{Suspend}$ to allow devices to suspend its own shut down.
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 one through four may be repeated until $t_{Suspend}$ expires before receiving a request to suspend the Device Shutdown process. Then the Execution stage is entered.

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. The Device mutes any synchronous outputs.
3. The Device unregisters itself from notification matrices in other devices, if any.
4. The Device unregisters its function blocks by sending an FBlockIDs.Status() with an empty FBlockIDList.
5. The NetworkMaster broadcasts the device's invalid function blocks.
6. The device can shut down its application but the NetBlock has to stay active.

3.2.4.3.2 Waking from Device Shutdown

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

WakeUp by PowerMaster

1. The PowerMaster sends Shutdown.Start(WakeFromDeviceShutdown).
2. The Device wakes its application.
3. The Device registers its own function blocks using FBlockIDs.Status(FBlockIDList).
4. When Network Master reports the new function blocks they can be used.

Internal Wakeup

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 function blocks using FBlockIDs.Status(FBlockIDList).
3. When Network Master reports the new function blocks they can be used.

3.2.4.3.3 Persistence of Device Shutdown

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

3.2.4.3.4 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).

3.2.5 Error Management

In the network the following errors may occur on the lowest 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 (Super- or sub-voltage, ring break, defect FOT 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 all-bypass, which means it “disappears” or “appears” as a new node.
- **Voltage Low:** The voltage of one or more devices is too low to maintain operation of the NetInterface.

For the handling of these errors, there are the following general rules:

- **No alert communication:** For keeping error management simple, robust and not error-prone, there is no communication in case of an error.
- **Local Handling Of Errors:** Every device is responsible to handle every recognized error locally. Only the NetworkMaster handles errors for the entire network.
- **Securing Synchronous Signals:** In opposite to the packet data area and the control channel, there is no data securing for the synchronous area. So data transported here, is sensitive for disturbances in case of errors. A device that recognizes an error, should immediately secure all output signals that depend on synchronous data transfer. This applies for instance to an audio amplifier, which has to mute its analog output signal (the one connected to the speakers). The synchronous 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).

3.2.5.1 Handling of Light Off

If a device recognizes at its input that light was switched off, it switches off its own output immediately. In case there is the need to wake the network again, it has to wait for $t_{Restart}$. If light was switched off without a ShutDown.Start (Execute), there may be two causes:

- Fatal error (voltage low, ring break), which is described below
- A device runs error handling (e.g., long unlock). In such a case the PowerMaster switches on light again, if the vehicle’s status requires it. So it wakes the network in the normal way. By that a re-initialization is done.

If light was switched off, it may be the case that it is switched on again after a short time. If the application would shut down immediately, some devices may 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 light 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 to be re-initialized.

3.2.5.2 Fatal Error

A “fatal error” is a kind of error that prevents the light from being handed on in the Ring. There are four possible reasons:

- A device (especially optical transmitter, optical transceiver, or a MOST Network Interface Controller) has no, or an insufficient distribution voltage.
- An optical receiver is defect.
- An optical transmitter is defect.
- The optical connection between transmitter and receiver is interrupted

3.2.5.2.1 Waking

If a fatal error occurs while an application tries to wake the network, “light on” does not propagate through the entire ring, and the Network Service in every device change to state NetInterfaceOff after t_{Config}. The waking application waits for $t_{Restart}$ and then tries again to wake the network. This will be repeated up to three times and then it suspends the waking. Only the PowerMaster tries to start up the network if required by the vehicle’s status.

3.2.5.2.2 Operation

If there is a fatal error during normal operation, “light 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).

3.2.5.3 Unlock

An unlock occurs when a timing Slave cannot lock onto the input signal of the PLL of the MOST Network Interface Controller, or if a Timing Master does not receive a comprehensible signal.

One cause for this may be that two Timing Masters in one ring work against each other. This case can be recognized only in the Timing Masters themselves.

Another cause can be that the optical signal at a node’s input is too weak, or a node opens or closes its all-bypass. Every node downstream from the location that caused the unlock, up to the Timing Master, recognizes the unlock. The nodes downstream of the Timing Master 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.

Reaction of the Network Service:
The Network Service of every device report an unlock immediately to the application by an Unlock event. In addition to that, the length of the unlock, and the occurrence of short unlocks is checked. If the network seems to be unstable due to long or frequent unlocks, an ErrorShutDown is performed by the Network Service. This is reported to the application with the help of a NetOff event. Following the context of its standard tasks, the PowerMaster tries to wake the network again after that.

Reaction on application level:
The following applies to all sinks in the system.
The application secures the synchronous signals. For example, an audio amplifier must mute as fast as possible (refer to section 3.8.1.4). After a lock is established again (recognized by the Network Service), the application restores its synchronous signals as fast as possible (e.g., de-mute). This must happen only if there is no Network Change Event, where a node has closed its all-bypass and therefore has left the network.

Reaction on NetworkMaster:

The NetworkMaster does not react on an unlock in a specific way. Both errors are stored in the error memory in the same way as in other devices.

3.2.5.4 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 All Bypass, i.e. 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]).

Disturbance on the Maximum Position Information can occur, e.g., from an unlock.

A NCE is recognized by the Network Service in every device.

If an additional node joined the network, the new node must be integrated on system level. Therefore SystemCommunicationInit must run (refer to chapter 3.3.1.1.2). In order to achieve that the NetworkMaster checks configuration again and broadcasts Configuration.Status.

If a node has left the network, the output signals that depend on synchronous data transfer must be secured immediately. 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.

[1] Typically up to 24 ms

3.2.5.5 Failure of a Function Block

It can be that single processes in a Device are hanging (but not the entire device), and that those processes need to be restarted. In case this failure stops an entire, or even several Function Blocks (FBlocks), the device has to un-register those function blocks in the Central Registry in the NetworkMaster. This is done through a notification of the new status of FBlockIDs sent to the NetworkMaster:

```
Device -> NM: NetBlock.RxTxLog.FBlockIDs.Status (FBlockIDList)
```

This tells, that only those function blocks contained in FBlockIDList are available. The NetworkMaster updates the Central Registry and broadcasts immediately after the reception of such an un-registration:

```
NM -> All: NetworkMaster.1.Configuration.Status   (Control=Invalid,
                                                    DeltaFBlockIDList)

Control              uns. Byte     0: NotOK
                                   1: OK
                                   2: Invalid
                                   3: New

DeltaFBlockIDList                  List of FBlockID.InstID
```

A detailed description of the handling of "Control = Invalid" and "Control = New" is to be found in sections 3.3.3.6 and 3.3.4.3.

DeltaFBlockIDList means the list of those function blocks that are invalid. Therefore, all applications have the required information and can terminate functions depending on the invalid function blocks.

If the failed process is ready (after being killed and re-initialized), the depending function blocks are registered again:

```
Device -> NM: NetBlock.RxTxLog.FBlockIDs.Status (FBlockIDList)
```

The NetworkMaster registers those function blocks in the Central Registry, and broadcasts immediately after having received the registration:

```
NM -> All: NetworkMaster.1.Configuration.Status (Control=New,
DeltaFBlockIDList)
```

Here, DeltaFBlockIDList means the list of the new function blocks.

Please note:
In case a device, that starts up fast, has single function blocks starting up relatively slow, the same mechanism of supplementary registration can be used. But the status message may not be sent before the NetworkMaster has asked the device.

When the Network Master reports Central Registry updates, the DeltaFBlockList must contain a maximum of five function blocks. This allows the message to be sent within a single telegram (11 Bytes max.). This limitation refers to such nodes, which can handle single telegrams only. In case more than five function blocks must be reported, several single telegrams are sent. Refer to sections 3.3.3.6.1 and 3.3.3.6.2.

3.2.5.6 "Hanging" of an Application

By implementing a watchdog in each device, a long "hanging" of the application should be avoided. This effect is reduced to a Network Change Event (Closing of all-bypass), and an eventual second Network Change Event (Opening of all-bypass).

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.

3.2.5.7 Failure of a Network Slave Device

If a device experiences an internal failure and recovers from that by restarting, all its function blocks must be unregistered in the Central Registry. When the Network Master announces the loss of the function blocks through a Configuration.Status message, the device must register its function blocks again.
The device must always wait until it is asked by the NetworkMaster, then return the empty list. The device must not communicate until it receives a Config.Status message. If the System State is OK (section 3.3.4.3.8), then the device must register its FBlocks.

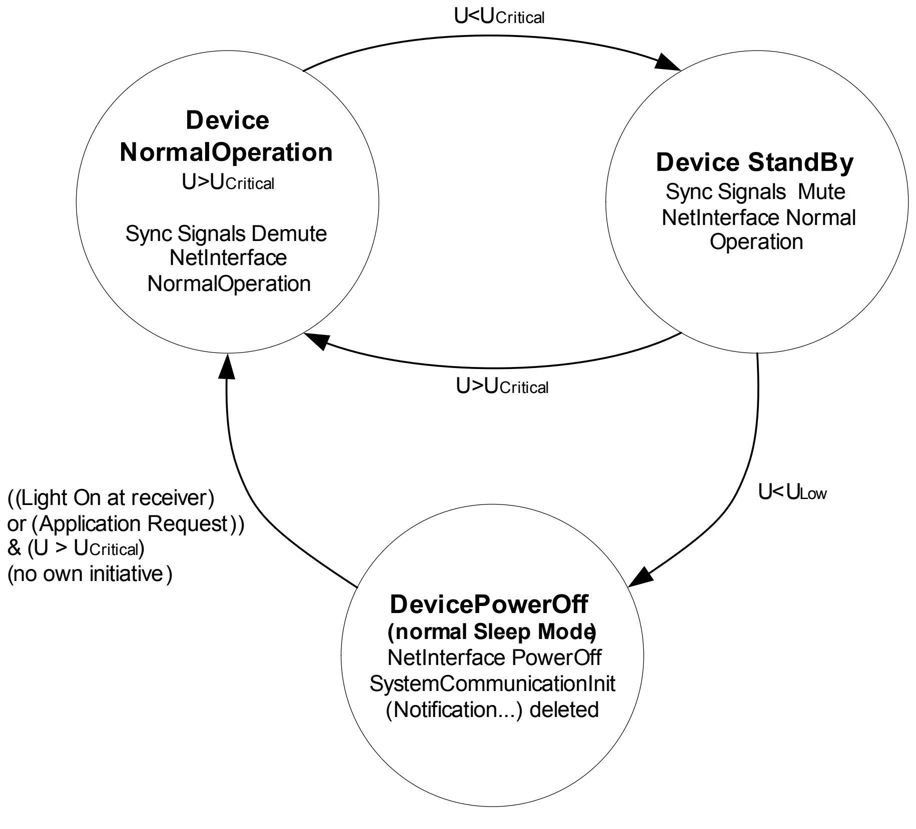
3.2.5.8 Low Voltage

The definition of voltage levels and more information can be found in section 4.7 on page 213. A too-low supply voltage does not inevitably occur in every device at the same time and in the same intensity. As already described, there are two limits regarding the supply voltage of a device:

Critical voltage $U_{Critical}$:
First, there is the limit at which the application will no longer work safely, but where communication is still possible. Since the application does not work any longer, the output signals that depend on synchronous data transfer must be secured. In case of a recovery, they can be restored immediately.

Low voltage U_{Low}:
There is a second limit, where even the NetInterface no longer works reliable, so even communication cannot be maintained.
If low voltage is reached the device is reset, then it switches off the light and switches to normal DevicePowerOff Mode, as if it was switched off. The device stays in DevicePowerOff mode, even if the supply voltage recovers. It is awakened either by "light on" at its input, or by the demand for communication from its own application. It changes to mode DeviceNormalOperation via the standard initialization process. The low voltage reset leads a device to normal behavior.

By opening all-bypass, the device indicates that it is joining the network. The other devices must then integrate it into the system via SystemCommunicationInit. Further signaling is not required here as well.


U<UCritical
Device NormalOperation
U>UCritical
Sync Signals Demute
NetInterface NormalOperation
Device StandBy
Sync Signals Mute
NetInterface Normal Operation
U>UCritical
((Light On at receiver) or (Application Request)) & (U > UCritical) (no own initiative)
U< ULow
DevicePowerOff
(normal Sleep Mode)
NetInterface PowerOff
SystemCommunicationInit (Notification...) deleted

[1]

Figure 3-17: Behavior of a device depending on supply voltage

[1] Note: It is up to the system integrator to decide whether an application is powered or not.

3.2.6 Over-Temperature Management

3.2.6.1 Introduction

Some components 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 recommendation is applicable to every device, which can monitor its own temperature and decide when to take appropriate action.
Different strategies are presented for the re-start of the system; they work independent from each other and can even be mixed within one system, if desired.

3.2.6.2 Levels of Temperature Alert

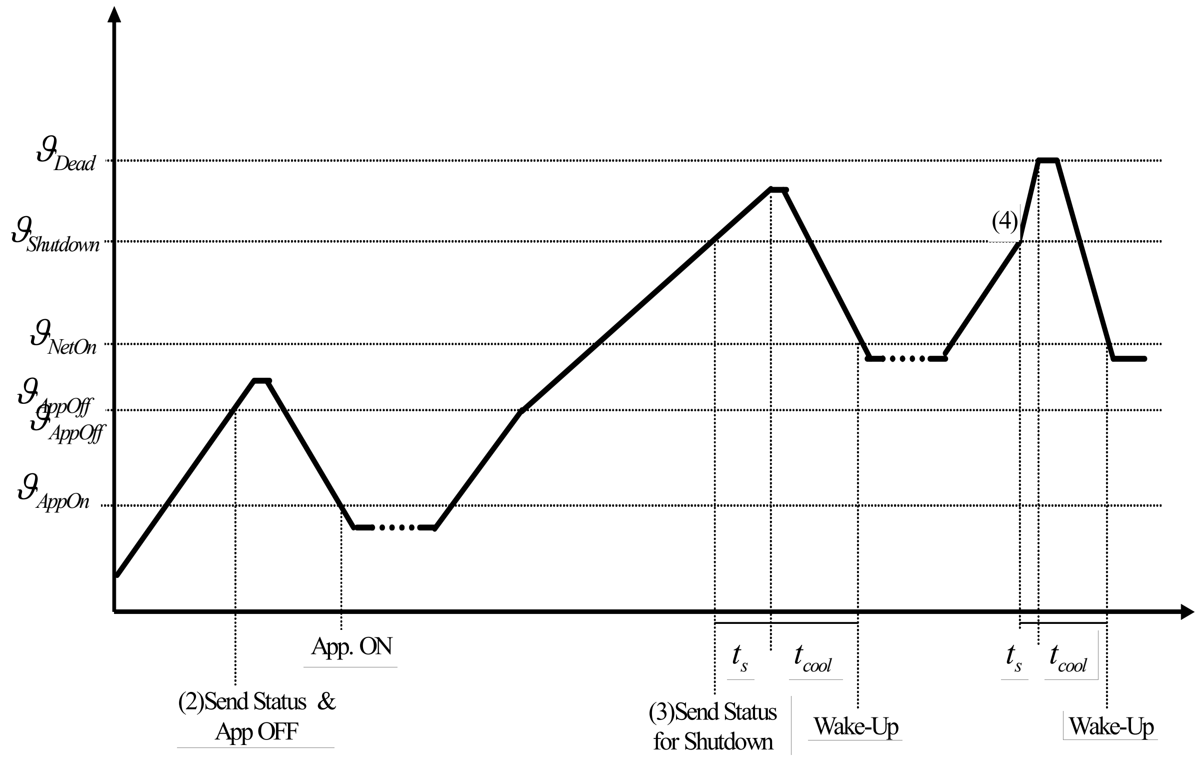


Figure 3-18: Alert Levels

Figure 3-18 shows three of the four different temperature alert levels that can be identified. Starting with the lowest one, they are:

1. Limited application functionality (not shown in the figure). 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.

2. Shutdown of individual applications. If, for example, a telephone unit becomes warmer than its maximum operating temperature (which is still below the maximum operating temperature for the FOT unit or other components needed for MOST functionality), the device could decide to shut down this specific application. The function block is then removed from the CentralRegistry by sending an updated NetBlock.FBlockIDs.Status message to the NetworkMaster.
3. If the temperature comes near the critical limit, the device should request a temperature shutdown from the PowerMaster. This is done by broadcasting NetBlock.Shutdown.Result with parameter 0x03 (Temperature Shutdown). The PowerMaster will then execute the standard shutdown procedure, please refer to section 3.2.4.2. Before that, it will set the AbilityToWake property of all devices except the one with the temperature problem to "Off".
4. If the critical temperature is finally reached (e.g., if the normal shutdown procedure does not finish due to a device constantly sending NetBlock.Shutdown.Result(Suspend)), an immediate shutdown is initiated by the device, which is in critical condition by simply switching the light off. Since the PowerMaster is aware of the over-temperature condition (because of having received the device's broadcast message before), it 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.

From these alert levels, only the following subset is mandatory: Point 4 as described above, and the broadcasting of the message NetBlock.Shutdown.Result(0x03) by the device as described under point 3. All other measures, including the standard shutdown procedure mentioned under point 3, are optional and can be used independently from each other.

3.2.6.3 Re-Start Behavior

After the system has been shutdown (see points 3 and 4 above), it has to stay off long enough for the device to cool down. The temperature should sink to a level that guarantees a reasonable amount of operation time when the system is back up again, i.e. it is not very useful to have a system that re-starts and remains in operational state for just a minute.
There are several ways to determine when and how the system shall be re-started. Like with the alert levels, they can be combined in several ways.

a) If the device is able to supervise its own cooling phase, it may wake up the ring when it has cooled down.
b) The PowerMaster may decide, after a while, to try a re-start. It simply wakes up the ring.
c) The PowerMaster could be triggered to re-start the ring upon user request.

Independent of those three re-start methods, the following is mandatory for the re-start procedure:

- If the device finds that it is still above the re-start temperature threshold, it broadcasts NetBlock.Shutdown.Result(0x03) again immediately after the NetOn state is reached. The PowerMaster shuts down the system again (without the standard procedure).
- If at re-start the NetworkMaster reaches the state "Configuration OK", the over-temperature condition of the system is over and the PowerMaster resets the AbilityToWake properties of all devices to their original state.

- A minimum time between re-start attempts of the system shall be guaranteed so the device has a chance to cool down. After all, a failing attempt to re-start the network will last no longer than approx. 150ms, so if the minimum interval between such attempts is e.g., one minute, the short phase of operation can be considered insignificant.

Note that all temperature levels (those for the alerts as well as those for re-start) are device-specific and are handled on a device-internal basis. No central component supervises the temperature of a device and decides for the device when it has to shutdown; this is completely at the device's discretion.

3.3 Network Management

Network Management is the process by which the Network Master ensures secure communication between applications over the MOST Network. This section describes the conditions that must be met by the Network Master and the Network Slaves 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 3.3.1 contains general descriptions of Network Management. Detailed requirements of the behavior of MOST devices regarding Network Management are described in sections 3.3.2 through 3.3.4. For more implementation specific information and examples refer to Appendix A.

3.3.1 General Description of Network Management

3.3.1.1 System Startup

This section describes the System Startup following the NetOn event.

3.3.1.1.1 Initialization of the Network

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

The Network Master 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 Network Slaves. This event prevents the same collisions to occur more than once.

The procedure by which the Network Master initializes the system depends on the availability of a valid logical node address and a Central Registry.

- If there is no valid logical node address available at System Startup the Network Master resets the network by setting the System State to NotOK before scanning the system.
- If there is a valid logical node address available at System Startup but no Central Registry the Network Master starts to scan the system.
- If there is a Central Registry available at System Startup the Network Master must verify that the Central Registry is still valid. If no mismatches are detected the Network Master sets the System State to OK, completing the network initialization. If any mismatch is detected the Network Master sets the System State to NotOK, clearing its Central Registry, before scanning the system again.

The Network Master will set the System State to NotOK whenever an error is caused by a Network Slave registration. The Network Master will continue to scan the system until there are no errors in Network Slave registrations and the System State is set to OK. A transition to System State OK indicates the completion of the network initialization.

3.3.1.1.2 Initialization on Application Level

After the Network Master 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, e.g., Notification is established, so the application of a device may register in the Notification Matrices of those function blocks 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 function blocks 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.

3.3.1.2 General Operation

3.3.1.2.1 Finding communication partners

When an application seeks a communication partner, that is, a function block; it requests the whereabouts of the function block from the Network Master. The requesting application receives the available InstIDs of the sought function block and the logical node addresses of the devices in which they reside. Alternatively, if a specific function block 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 Network Slave 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 Network Master sets the System State to NotOK.

3.3.1.2.2 Network Monitoring

The Network Master monitors the system for changes and errors. When a Network Change Event is detected, the Network Master must find out if a device has entered or left the network. It scans the network and reports any new information to all Network Slaves 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 Network Master may be instructed to scan the system at any time by the application.

3.3.1.2.3 Dynamic Function Block Registrations

It may happen that devices activate and deactivate function blocks at any time; these changes have to be reported to the Network Master. The Network Master then updates the Central Registry and informs all Network Slaves. This also applies for devices experiencing failures. If a device fails, an NCE is detected by the Network Master, which then scans the system.

3.3.2 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 Network Master builds and maintains the Central Registry as well as distributes the System State to all Network Slaves.

The Network Master builds the Central Registry by collecting logical node addresses and function block configuration from all Network Slaves. 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 Network Master distributes the System State of the network to the Network Slaves by broadcasting Configuration.Status() messages. The state diagram in Figure 3-19 shows the System States and which events affect the states.

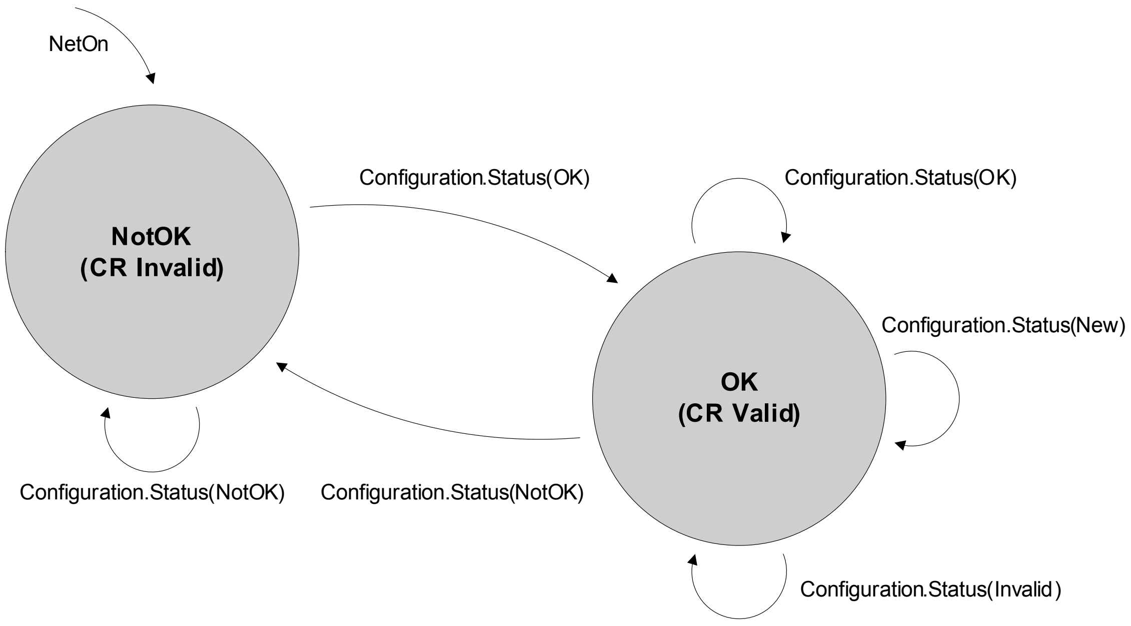

NetOn
NotOK
(CR Invalid)
Configuration.Status(OK)
Configuration.Status(OK)
Configuration.Status(New)
OK
(CR Valid)
Configuration.Status(NotOK)
Configuration.Status(NotOK)
Configuration.Status(Invalid)

Figure 3-19: States of the network are shown, as well as the status of the Central Registry

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

Sections 3.3.3 Network Master and 3.3.4 Network Slave describe device specific behavior in the different System States as well as making transitions between states.

3.3.2.1 System State NotOK

System State NotOK is always entered after a NetOn 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-dependant basis. The system can fall back into System State NotOK at any time by declaration of the Network Master.

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. The 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 Network Master - Erroneous registration by network slave.	Network Configuration Reset: - Clear Central Registry - Clear Decentral Registries - Recalculate NodeAddress
OK	SystemState OK	- Central Registry verified	- Network configuration available in Central Registry - Set up Decentral Registries, where necessary. - (Re-) initialize applications.

Table 3-10: Events in System State NotOK (refer to Figure 3-19)

3.3.2.2 System State OK

While in System State OK, the Central Registry is valid. So the exact set of function blocks 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 control and asynchronous channels. All the dynamic communication on application level within the distributed system should be done only in System State OK.

Event Configuration.Status()	Transition to	Cause	Effect
NotOK	SystemState NotOK	- Erroneous registration by network slave.	Network Configuration Reset: - Clear Central Registry - Clear Decentral Registries - Recalculate NodeAddress
OK	No transition	- Large update to the Central Registry	- Clear Decentral Registries
New	No transition	- New FBlocks are available	- Notify application
Invalid	No transition	- FBlocks were removed	- Notify application

Table 3-11: Events in System State OK (refer to Figure 3-19)

Table 3-11 shows the effects of a Configuration.Status(NotOK) event in System State OK from a Network Management point of view. In addition, the following tasks have to be performed by all devices:

- Empty notification lists.
- Destroy all windows on LongArrays.
- Every function block containing synchronous sinks: set the Mute property to "ON" for all sinks and disconnect them.
- Every function block containing synchronous sources: All sources must route zeroes (signal mute) to its channels for a time $t_{CleanChannels}$. After time $t_{CleanChannels}$ all synchronous sources must de-allocate the channels they have allocated and stop routing data to the network.
- The Connection Manager must delete its SyncConnectionTable.

3.3.3 Network Master

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

The Network Master controls the System State and administrates the Central Registry. The Network Master monitors the network for certain events and continuously manages incoming information from Network Slaves about their current function block configuration and whenever necessary informs all Network Slaves about updates to the Central Registry.

3.3.3.1 Setting the System State

The Network Master distributes the System State by broadcasting Configuration.Status messages. More information about the different System States and Configuration.Status messages is available in section 3.3.2.

3.3.3.1.1 Setting the System State to OK

By setting the System State to OK, the Network Master confirms the validity of the Central Registry. Therefore, before setting the System State to OK, the Network Master must make sure that all functional addresses are unique in the system (section 2.3.2.3).

The Network Master 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.

3.3.3.1.2 Setting the System State to NotOK (Network Reset)

By broadcasting Configuration.Status(NotOK), the Network Master resets the system (from a network point of view). Note that this does not necessarily imply a state change, as described in section 3.3.2.

The Network Master must do the following when setting the System State to NotOK:

1. Broadcast Configuration.Status(NotOK).
2. Clear the Central Registry.
3. Derive and set the new logical node address (section 3.4.1).
4. Wait a time $t_{WaitBeforeScan}$ after Configuration.Status(NotOk) was broadcasted and perform a System Scan (section 3.3.3.4).

3.3.3.2 Central Registry

The Network Master generates the Central Registry during the initialization of the network and it continues to administrate it until Network Shutdown (section 3.2.4.2). The Central Registry is an image of the physical and logical system configuration. It contains the logical node address and the respective function blocks of each device:

Rx/TxLog	Rx/TxPos	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 3-12: Example of a Central Registry

3.3.3.2.1 Purpose

The Central Registry is used when the Network Master checks the system configuration and when devices are searching for communication partners or their physical addresses.

3.3.3.2.2 Contents

The Central Registry must contain the logical node address and the respective functional addresses (combination of FBlocks and InstIDs) of the function blocks in each responding MOST device. This information must be made available to all Network Slaves.

3.3.3.2.3 Persistence of the Central Registry

It is optional to store the Central Registry between system runs; however, the Network Master may store the Central Registry only after Network Shutdown in its proper form (section 3.2.4.2).

3.3.3.2.4 Responsibility

Any new information gained regarding the system configuration must be entered into the Central Registry and distributed to all Network Slaves as described in section 3.3.3.6.

The Network Master must only start supervising and store errors for those Network Slaves, that have answered requests and which are registered in the Central Registry.

3.3.3.2.5 Responding to Requests for Information from the Central Registry

The Network Master must respond to requests for CentralRegistry.Get() from the Network Slaves while the System State is OK. This is described in section 2.3.7.

3.3.3.2.6 Secondary Nodes

In the Central Registry it must be marked, which node is the Primary Node to a certain Secondary Node. The Central Registry must be sorted in such a way, that the entry of a Secondary Node directly succeeds the entry of the Primary Node.

For completing the entries of Secondary Nodes in the Central Registry, each Secondary Node is registered with a single function block having FBlockID.InstID = 0xFC.0.

Note that there may be more than one secondary node in a system. This is the sole exception to the rule of unambiguousness of the entries in the Central Registry. Secondary Nodes are described in section 3.2.3.

3.3.3.3 Specific Behavior During System Startup

After the NetOn event the Network Master must initialize the system. This process depends on the availability of a valid[1] logical node address and a Central Registry.

3.3.3.3.1 Valid Logical Node Address Not Available

If the Network Master does not have a valid logical node address available at System Startup it assumes that the entire system must be re-initialized. The Network Master must set the System State to NotOK (section 3.3.3.1.2) and then start a System Scan (section 3.3.3.4).

3.3.3.3.2 Valid Logical Node Address Available but No Central Registry

If the Network Master has a valid logical node address but no Central Registry available at System Startup, it must restore its logical node address and then start a System Scan (section 3.3.3.4).

3.3.3.3.3 Valid Logical Node Address and a Central Registry Available

If the Network Master has a valid logical node address and a Central Registry available at System Startup, it must restore its logical node address and then verify that the Central Registry is still valid. This procedure is referred to as a Verification Scan (section 3.3.3.8).

3.3.3.3.4 Stable Network

The NetworkMaster should wait a time $t_{WaitBeforeScan}$ before scanning the system for the first time. This latency time allows the system to stabilize after NetOn event. This latency time must not exceed $t_{WaitAfterNCE}$.

[1] A valid logical node address is any address within the dynamic or static address ranges as defined in section 3.1.5.1.[0]

3.3.3.4 Scanning the System (System Scan)

The Network Master 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 Network Master scans the system by requesting the function block configuration of each device. The responses from the Network Slaves are interpreted as described in section 3.3.3.5. Any information gained concerning the configuration of the network must be written to the Central Registry and reported to all Network Slaves as described in section 3.3.3.6.

3.3.3.4.1 Configuration Request Description

During a network scan, the Network Master requests NetBlock.FBlockIDs.Get() from each Network Slave.

3.3.3.4.2 Addressing

The Network Master scans the system by node position addressing. The logical node address of the requested Network Slave is contained in the response message.

3.3.3.4.3 Non Responding Network Slaves

The Network Master must wait until the expiration of $t_{WaitForAnswer}$ for a reply from a Network Slave. The Network Master must send another request to the Network Slave as described in section 3.3.3.4.4.

3.3.3.4.4 Retries of Non Responding Network Slaves

When a Network Slave does not respond to a request, the Network Master must try again after $t_{DelayCfgRequest1}$ or $t_{DelayCfgRequest2}$. $t_{DelayCfgRequest1}$ is used for the first 20 request attempts after entering NetInterfaceNormalOperation, after that $t_{DelayCfgRequest2}$ is used.

Refer to section 3.9 for more information about timers.

3.3.3.4.5 Network Slave Continuous cause for System State NotOK

The Network Master should ignore a Network Slave which has caused the Network Master to broadcast Configuration.Status(NotOK) three times in succession. The Network Master should ignore the Network Slave until the next NCE or the next System Startup.

3.3.3.4.6 Duration of System Scanning

The Network Master must continue to scan the system until all Network Slaves have answered the requests. Refer also to section 3.3.3.4.4.

3.3.3.4.7 Reporting the Results of a System Scan without Errors

The Network Master 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 function block configuration of one or more Network Slaves. Refer to sections 3.3.3.1 and 3.3.3.6.

3.3.3.5 Invalid Registration Descriptions

The Network Master 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 Network Master.

3.3.3.5.1 Un-initialized Logical Node Address

If any Network Slave, at any time, registers an un-initialized logical node address (section 3.1.5.1), the Network Master must set the System State to NotOK (section 3.3.3.1.2), interrupting any ongoing System Scan.

3.3.3.5.2 Invalid Logical Node Address

When the Network Master receives a registration from a Network Slave, in which its logical node address is outside of the specified address range, the Network Master must set the System State to NotOK (section 3.3.3.1.2), interrupting any ongoing System Scan.

Refer to section 3.4.1 for more information about the valid address range.

3.3.3.5.3 Duplicate Logical Node Addresses

When the Network Master receives a registration from a Network Slave, in which its logical node address has already been registered by another Network Slave, the Network Master must set the System State to NotOK (section 3.3.3.1.2), interrupting any ongoing System Scan.

3.3.3.5.4 Duplicate InstID Registrations

The Network Master is responsible for the uniqueness of functional addresses (combination of FBlockIDs and InstID) within the entire system. The Network Master must try to resolve the issue of two or more Network Slaves registering identical functional addresses.

The Network Master decides a new InstID for the last registered function block. It then sets the new InstID in the corresponding Network Slave. If the new InstID was accepted by the Network Slave, the Network Master enters the new value into the Central Registry. The Network Master must inform all Network Slave as described in section 3.3.3.6 or by ultimately setting the System State to OK.

If the request to change the InstID of a conflicting function block is not successful, the Network Master must not include the function block into the Central Registry.

3.3.3.5.5 Error Response

A Network Slave that answers a request from the Network Master with an error must be treated as a non-responding Network Slave (section 3.3.3.4.4). The exception to this rule is the correct response of a Secondary Node (section 3.2.3).

3.3.3.6 Updates to the Central Registry

The Network Master must inform all Network Slaves about changes of the system configuration. This information may become available during a System Scan or as Network Slaves make additional registrations, which are not requested by the Network Master.

This section describes how the Network Master handles changes to the system configuration while in System State OK.

3.3.3.6.1 Disappearing Function Blocks in System State OK

If the Network Master receives a registration from a Network Slave in which there is one or more function blocks missing compared to the last registration from the same Network Slave, the Network Master must update the Central Registry and inform all Network Slaves about the missing function blocks. This is done by broadcasting:

Configuration.Status(Invalid, DeltaFBlockIDList)

The DeltaFBlockIDList parameter is a list of the previously registered but now invalid function blocks. The DeltaFBlockIDList must only contain five function blocks, if there are more than five invalid function blocks, several single messages must be sent.

When one or more function blocks have disappeared the Network Master should inform all Network Slaves about the missing function blocks as quickly as possible, even if this information is gained while scanning the network.

3.3.3.6.2 Appearing Function Blocks in System State OK

If the Network Master receives a registration from a Network Slave in which there is one or more additional function blocks compared to the last registration from the same Network Slave, the Network Master must update the Central Registry and inform all Network Slaves about the new function block. This is done by broadcasting:

Configuration.Status(New, DeltaFBlockIDList)

The DeltaFBlockIDList parameter is a list of the new function blocks. The DeltaFBlockIDList must only contain five function blocks, if there are more than five new function blocks, several single messages must be sent.

If this information is gained while scanning the network, the Network Master may continue to scan the system before it informs all Network Slaves.

3.3.3.6.3 System scan without any change in Central Registry

The Network Master shall broadcast a Configuration.Status(New) with an empty list when a network scan that was triggered by an NCE did not detect any changes to the registry.

3.3.3.6.4 Large Updates to the Central Registry in System State OK

If the Network Master receives registrations, which result in large updates to the Central Registry, it either broadcasts the respective sequence of New and Invalid messages or just a Configuration.Status(OK). Both methods indicate to the Network Slaves that there is a new, updated Central Registry available. The latter method requires the Network Slaves to fetch the differences themselves.

3.3.3.6.5 Non-responding Devices in System State OK

If a Network Slave, which has registered in the Central Registry since startup, does not respond to the request before $t_{WaitForAnswer}$ expires, the Network Master removes the Network Slave from the Central Registry and informs all Network Slaves as described in section 3.3.3.6.1.

3.3.3.7 Miscellaneous Network Master Requirements

3.3.3.7.1 Network Change Event (NCE)

When an NCE is detected, the Network Master must start a complete System Scan (section 3.3.3.4) after $t_{WaitAfterNCE}$. This also applies to a Verification Scan at System Startup (section 3.3.3.3). Any scan in progress when the NCE is detected must be interrupted and restarted.

3.3.3.7.2 Positioning of the Function Block NetworkMaster in the MOST Network

The NetworkMaster function block must be located in a MOST device with a node position address such that it fulfills the requirement of $t_{MPRDelay}$ (refer to section 3.9). The NetworkMaster function block is normally located in the same MOST device as the TimingMaster.

3.3.3.8 Verifying the Central Registry at System Startup (Verification Scan)

The Verification Scan is only performed if there is a Central Registry available at System Startup. The Network Master performs a Verification Scan to verify that the Central Registry is still valid, that is, it is a valid representation of the current system configuration. The Verification Scan is basically a normal System Scan (section 3.3.3.4) with the difference that all responses from the Network Slaves have to match the Central Registry exactly.

The Verification Scan finishes and the System State is set to OK when the Network Master has requested all Network Slaves (ignoring nodes that do not respond) and the received registrations are without differences to the Central Registry (logical node address and contained function blocks).

The Verification Scan finishes and the System State is set to NotOK if any registration does not match the buffered Central Registry. The Network Master must perform a normal System Scan after setting the System State to NotOK.

Note that Missing Devices are tolerated during a Verification Scan (section 3.3.3.8.1).

3.3.3.8.1 Missing Devices

A Missing Device is a non-responding Network Slave (section 3.3.3.4.3) that is available in the buffered Central Registry but has not replied to any requests from the Network Master since startup; therefore, its existence in the system cannot be confirmed.

A Network Slave is considered missing if it does not respond to a request from the Network Master before the expiration of $t_{WaitForAnswer}$ during a Verification Scan. When the Network Slave registers correctly it is no longer considered missing and is entered in the Central Registry.

3.3.3.8.1.1 Requesting Missing Devices

If there is a Missing Device in the system, the Network Master must try to request the Missing Device again after $t_{DelayCfgRequest1}$ or $t_{DelayCfgRequest2}$ has expired. $t_{DelayCfgRequest1}$ is used for the first 20 request attempts after entering NetInterfaceNormalOperation, after that $t_{DelayCfgRequest2}$ is used. The Network Master must continue to do so until the Network Slave has responded.

Refer to section 3.9 for more information about timers.

3.3.3.8.1.2 Matching Response of Missing Device

When a Missing Device registers and the registration matches the registration in the buffered Central Registry the Network Master must distribute this information as a change in the system configuration (section 3.3.3.6).

3.3.3.8.1.3 Non-matching Response of Missing Device

When a Missing Device makes a registration and the registration does not match the Central Registry exactly, the Network Master must change the previous entry and distribute the new information as a change in the system configuration (section 3.3.3.6).

3.3.3.8.1.4 Receiving a Central Registry Request for a Missing Function Block

The Network Master must only search the verified entries in the Central Registry if a Network Slave requests the whereabouts of a communication partner (section 3.3.1.2.1). This means that the function blocks of a Missing Device must not be reported, instead an error code must be returned (see section 2.3.7).

3.3.4 Network Slave

All devices that do not contain the NetworkMaster function block are called Network Slaves. A Network Slave must keep the Network Master informed about its current function block configuration.

3.3.4.1 Decentral Registry

Devices that control other devices should build a Decentral Registry in which it registers its communication partners. 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 3-13: Example of a Decentral Registry

3.3.4.1.1 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.

3.3.4.1.2 Updating the Decentral Registry

The function block entries stored in the Decentral Registry must match the entries of the respective function block in the Central Registry. When the Network Master informs of updates to the Central Registry; the Decentral Registry must be updated accordingly for the registered function blocks.

3.3.4.1.3 Deleting the Decentral Registry

The Decentral Registry must be cleared when the Network Master broadcasts Configuration.Status(NotOK) and whenever the device is removed from power.

3.3.4.1.4 Persistence of the Decentral Registry

It is optional to store the Decentral Registry between system runs; however, a Network Slave may store the Decentral Registry only after Network Shutdown in its proper form (section 3.2.4.2).

3.3.4.2 Specific Startup Behavior

Following the NetOn event the Network Slave initializes its logical node address and services requests from the Network Master.

3.3.4.2.1 Behavior When a Valid Logical Node Address is Not Available at System Startup

If the Network Slave does not have valid[1] logical node address available at System Startup, it must set its logical node address to the value of an uninitialized logical node address (section 3.1.5.1) and services requests from the Network Master until the Network Master sets the System State.

3.3.4.2.2 Behavior When a Valid Logical Node Address is Available at System Startup

If the Network Slave has a valid[1] logical node address stored from the previous system run, it uses that logical node address and services requests from the Network Master until the Network Master sets the System State.

3.3.4.2.3 Deriving the Logical Node Address of the Network Master

The logical node address of the Network Master must be derived from the Configuration.Status message at each System Startup.

[1] A valid logical node address is any address within the dynamic or static address ranges as defined in section 3.1.5.1.

3.3.4.3 Normal Operation of the Network Slave

3.3.4.3.1 Behavior in System State OK

A Network Slave may communicate freely while the System State is OK.

3.3.4.3.2 Behavior in System State NotOK

While the System State is NotOK a Network Slave 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-dependant basis.

A Network Slave must not send a NetBlock.FBlockIDs.Status(FBlockIDList) message in System State NotOK without being requested explicitly by the NetworkMaster.

3.3.4.3.3 Responding to Configuration Requests by the Network Master

The Network Slave responds to requests for function block configuration from the Network Master at all times, regardless of the current System State. The response must be sent before the expiration of t_{Answer}.

The Network Slave must report all function blocks that are currently active from a network point of view. The Network Slave must not include NetBlock nor function block EnhancedTestability when reporting its function block configuration. If the Network Slave does not have any active function blocks it must respond with an empty FBlockIDList.

3.3.4.3.4 Reporting Configuration Changes to the Network Master

When the function block configuration of a Network Slave changes, it must report this change to the Network Master; however, it must not do so if the current System State is NotOK (section 3.3.4.3.2).

3.3.4.3.5 Failure of a Function Block in a Network Slave

This behavior is described in section 3.2.5.5.

3.3.4.3.6 Failure of a Network Slave Device

This behavior is described in section 3.2.5.7.

3.3.4.3.7 Unknown System State

If the Network Slave does not know the current System State, it must assume that the System State is NotOK. For determining of system state refer to section 3.3.4.3.8.

3.3.4.3.8 Determining the System State

The current System State must be determined from the Configuration.Status message, which is broadcasted by the Network Master. The Config.Status (NotOK) implies the system status being NotOK. All other Config.Status messages imply the system state is OK e.g., Config.Status(Ok/New/Invalid/New,<empty>/Invalid,<empty>).

3.3.4.3.9 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 3.3.4.1).

3.3.4.3.10 Reaction to Configuration.Status(OK) When in System State NotOK

When the Network Master sets the System State to OK the Network Slave:

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 3.3.2.

3.3.4.3.11 Reaction to Configuration.Status(OK) When in System State OK

The Network Master sends this message to inform all Network Slaves that there is a large update to the Central Registry. All Network Slaves must:

1. Clear any Decentral Registry.
2. Rebuild a Decentral Registry when necessary.
3. The application must secure all synchronous data associated with any disappearing function blocks.

The System State remains in OK. For additional information, refer to section 3.3.2.

3.3.4.3.12 Reaction to Configuration.Status(NotOK) when in System State NotOK

The Network Master sends this message to reset all Network Slaves from a network point of view. All Network Slaves must:

1. Clear any Decentral Registry.
2. Derive and set the new logical node address (section 3.4.1).

The System State remains in NotOK. For additional information, refer to section 3.3.2.

3.3.4.3.13 Reaction to Configuration.Status(NotOK) When in System State OK

When the Network Master sets the System State to NotOK the Network Slave must:

1. Clear any Decentral Registry.
2. Derive and set the new logical node address (section 3.4.1).
3. Empty notification lists.
4. Destroy all windows on LongArrays.
5. Every function block containing synchronous sinks: set the Mute property to "ON" for all sinks and disconnect them.
6. Every function block containing synchronous sources: All sources must route zeroes (signal mute) to its channels for a time $t_{CleanChannels}$. After time $t_{CleanChannels}$ all synchronous sources must de-allocate the channels they have allocated and stop routing data to the network.
7. Service requests from the Network Master while waiting for the System State to be set to OK.

The System State is set to NotOK. For additional information, refer to section 3.3.2.

3.3.4.3.14 Reaction to Configuration.Status(New)

One or more function blocks have entered the system and the Central Registry has been updated with the function blocks supplied in the message. These function blocks 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 3.3.2.

3.3.4.3.15 Reaction to Configuration.Status(Invalid)

One or more function blocks have left the system and the function blocks supplied in the message have been removed from the Central Registry. These function blocks have to be removed from the Decentral Registry if entered. The application must secure any synchronous data associated with the disappearing function block.

This message is only sent in System State OK. The System State remains in OK. For additional information, refer to section 3.3.2.

3.4 Accessing Control Channel

3.4.1 Addressing

In a MOST network, nodes in a ring are addressed. The MOST Network Interface Controller provides five different types of addresses, which are described below.

- **Internal Node Communication Address**
  This address is reserved for internal communication in a node

- **Node Position Address (*Rx/TxPos*)**
  The Node Position Address is made up by the physical position of the MOST Network Interface Controller. The Node Position Address is called RxPos for a receiving node, and TxPos for a transmitting node.

- **Logical Node Address (*Rx/TxLog*)**
  User definable address. It must be unique in the system, and is called RxLog for receiving nodes and TxLog for transmitting nodes.

- **Group address**
  Provides access to a group of devices.

- **Broadcast address**
  All devices.

Addressing is done in the following way:

Node Position Address:

A node position address is unique by definition, but it has the disadvantage that the receiving MOST Network Interface Controller does not report the sender's node position address, but the sender's logical node address. This happens only when using node position addressing. For this reason, node position addressing is not used under normal operation conditions. It is used by the NetworkMaster only for administrative tasks, such as 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:

Rx/TxPos = 0x0400 + Pos

Pos = 0 for Timing Master
Pos = 1 for first device in ring...

Logical Node Address:

Logical node addressing is used by all nodes to address a single node. The section below describes the default procedure for assigning logical node addresses.

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:

Rx/TxLog = 0x0100 + Pos

The device containing the Timing Master is located at physical position zero; it will have the logical node address 0x0100. A device at position five in the ring will have address 0x0105.

Another approach is to assign certain address ranges with respect to the functionality of devices. That 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.

The logical node address can be requested from the function **NodeAddress** in the NetBlock.

The same logical node address must be used between two successive system runs unless the device is removed from power. If the device is removed from power it is optional to store the logical node address. After first power up, the logical node address is normally set to 0xFFFF (refer to sections 3.3.3.3 and 3.3.4.2) but it may also be set to a predefined system specific value.

Group Address:

The group address can be requested from function **GroupAddress** in the NetBlock, it can be modified using this function if required. The default procedure for deriving a group address is to take the FBlockID of the function block that is most characteristic for the device:

GroupAddress = 0x0300 + FBlockID

The function block (FBlockID) that is reported first in case of a request for the FBlockIDs is typically the most descriptive for the device.

Another approach for example, is that the system integrator may choose to use a hard coded group address for the whole system, i.e. that each device comes up with the same group address.

Groups can be built dynamically by modifying group addresses.

The group address is stored in unbuffered RAM, so it is lost if the device loses power for some time. If the device stays powered, the group address is kept. In case power gets lost, the default value (FBlockID) must be restored.

Broadcast Address:

Broadcast addressing requires a great deal of system resources and therefore should be used for administrative tasks only.

FUNCTIONS				
FktID	**OPType**	**Sender**	**Receiver**	**Explanation**
NodePositionAddress	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

Table 3-14: Functions in NetBlock that handle addresses

3.4.2 Assigning Priority Levels

Despite the high capacity of the control channel, temporary overload situations are possible, for example, during system initialization. Nevertheless, it must be possible to send important messages in that case. To do this, a fair arbitration mechanism is implemented in the MOST Network Interface Controller. For each control message a priority level can be assigned (range 0x00, …, 0x0F with value 0x00 = lowest priority).

3.4.3 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. Registers specify 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.

3.4.4 High Level Retries

High Level Retries are not planned at this time, since the expenditure in software development would be too great for the expected results. All devices have to safeguard the ability to accept messages within the interval of time given by the low level retries.

3.4.5 Basics for Automatic Adding of Physical Address

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

One way is when the application answers a request. In this case it already has the DeviceID of the target node because it was reported during the request. The other way is when the application is sending a protocol and does not know the DeviceID of the target. In this case 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 3.3.3.2 and 3.3.4.1 for information regarding the Central Registry and the Decentral Registries respectively.

Please note:
When seeking the logical node address of a communication partner, a device performs the following flow:

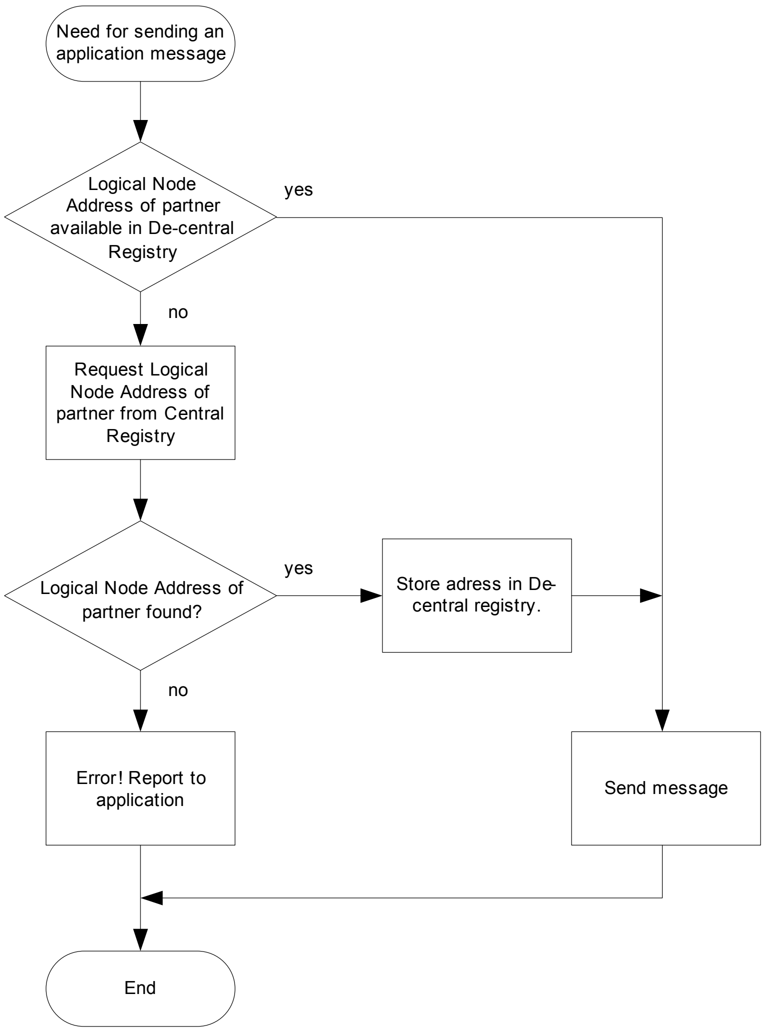

Need for sending an application message
Logical Node Address of partner available in De-central Registry
yes
no
Request Logical Node Address of partner from Central Registry
Logical Node Address of partner found?
yes
Store adress in De-central registry.
no
Error! Report to application
Send message
End

Figure 3-20: Seeking the logical address of a communication partner

3.4.6 Handling Overload in a Message Sink

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). The error is stored as Error_NAK.

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 data sink. A simple mechanism may look like this:

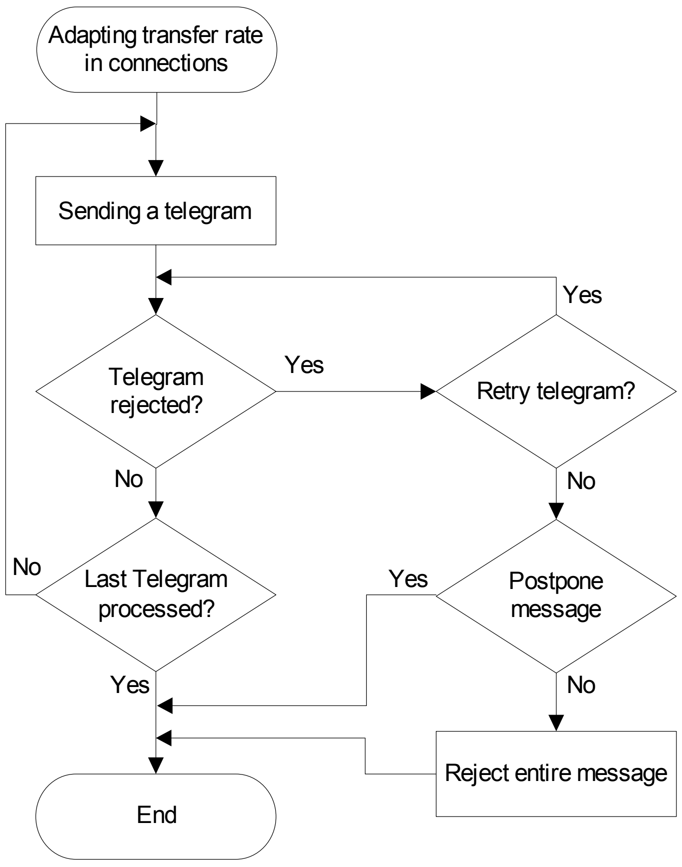


Figure 3-21: Possible mechanism to adapt transfer rates to the speed of a data sink

It is assumed here, that errors due to incorrect address, or CRC error 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, e.g., because it is no longer relevant.

3.) The entire message can be postponed, i.e., sent later.

3.4.7 MOST Message Services

3.4.7.1 Control Message Service

Via the MOST Network Interface Controller, MOST telegrams can be sent and received which consist of a sender or receiver address respectively (Rx/TxAdr), and a maximum number of 17 data Bytes.

Data area of MOST Network Interface Controller = 17 Byte

16 Bit										
Rx/TxAdr										

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.

3.4.7.2 Application Message Service (AMS) and Application Protocols

MOST Network Service provides three different types of transmissions via the control channel. Two of them are mandatory for each device:

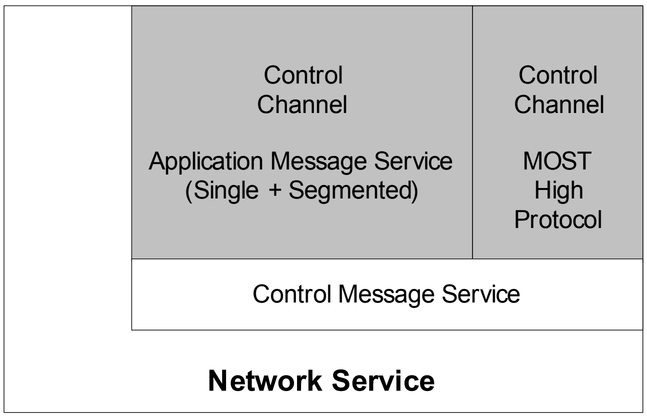


Figure 3-22: Network Service: Services for control channel

- **Single Transfer:** Data packets up to 12 Bytes are transmitted in a single telegram.
- **Segmented Transfer:** Commands and status messages with a length greater than 12 Bytes are transported by multiple telegrams. These segmented transfers have a maximum length of 65536 Bytes.

Single as well as segmented transfers are based on the application message service (AMS), which is mandatory for all MOST Network Services. In addition to that, a third transmission procedure is defined:

- **MOST High Protocol:** 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.

As already described in section 2.3.2 on page 38, protocols of the following type must be transmitted:

```
DeviceID.FBlockID.InstID.FktID.OPType.Length (Parameter)
```

The application message service (AMS), is based on the control message service (CMS). MOST telegrams transport application protocols. Each telegram is divided up as follows:

Data area of MOST Network Interface Controller = 17 Byte

16 Bit	8 Bit	8 Bit	12 Bit	4 Bit	4 Bit	4 Bit	8 Bit	8 Bit		8 Bit
DeviceID	FBlock ID	Inst. ID	Fkt ID	OP Type	Tel ID	Tel Len	Data 0	Data 1	...	Data 11

The parts of the application protocol are cross-hatched. The length of the application protocol is not transmitted directly. It has no meaning on telegram level, since several telegrams may be required to transport one protocol. Nevertheless, length is transmitted indirectly via TelLen and MsgCnt and must be restored on the receiver's side.

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). Telegram Segmented Transfer	2	0x(n-1)
Last telegram Segmented Transfer	3	0xn
MOST High Protocol User data	8	
MOST High Protocol Control data	9	

TelLen: = 0...12 specifies the length of the data field, i.e., the number of Bytes after TelLen; 0 means no data Byte

Data 0-Data 11: Data Bytes

3.5 Handling Synchronous Data

3.5.1 MOST Network Service API

The MOST Network Interface Controller provides mechanisms for administrating synchronous channels. In addition to that, the Network Service provides an API to simplify the use of those mechanisms.

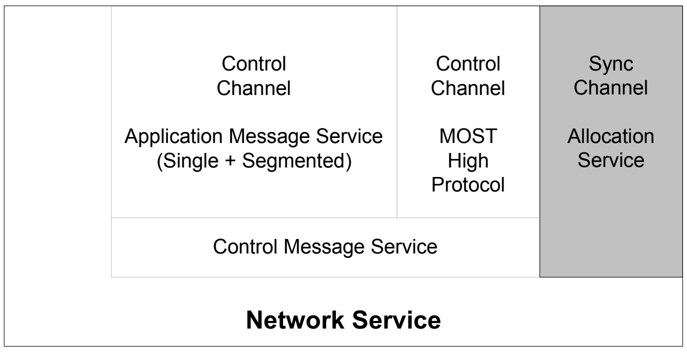


Figure 3-23: Network Service for the synchronous channel

On the network, 60 Bytes for synchronous and asynchronous transport are available per frame. A certain number of these channels can be used for synchronous data transfer. Here, several channels can be clustered to a synchronous connection for an application. Channels are grouped together into groups of maximum 8 channels. Every group is then assigned a Connection Label, which is the number of the lowest channel in the group. Using connection labels makes it easier to handle connections spanning several groups.

Access to the channels within a device (putting data onto the channels, or getting data from the channels) is done through the source data ports of the MOST Network Interface Controller in several different modes. Connecting the source ports with the channels is controlled via the Routing Engine (RE).

3.5.2 Function Block Functions

On the application level, different basic functions in sources and sinks are realized, which serve the administration of synchronous connections. They themselves access the routines of the Network Service. The function block 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 a function block.
- Functions that deal specifically with only sources or sinks within a function block.

3.5.2.1 NetBlock

NetBlock provides the following method to the network:

- **SourceHandles**
  A controller can request information of which function block is using a specific connection. To get the information it asks:

  ```
  Controller -> Slave: NetBlock.Pos.SourceHandles.Get (Handle)
  ```

  Since there can be several function blocks in a device using the same connection, the answer is:

  ```
  Slave -> Controller: NetBlock.Pos.SourceHandles.Status
                                                     (Handle, FBlockID.InstID,
                                                      Handle, FBlockID.InstID,
                                                      ...)
  ```

  In case the handle is not used, the following error is reported:

  ```
  Slave -> Controller: NetBlock.Pos.SourceHandles.Error (0x07, 0x01, Handle)
  ```

  If the controller specifies 0xFF as handle, it gets the handles of all connections used in the device, and the IDs of the function blocks using them.

  Please note: the SourceHandles function should only be used for debugging purposes.

3.5.2.2 General Source / Sink Information

A function block containing a source or sink provides the following function:

- **SyncDataInfo**
  Function SyncDataInfo can be used to get information of how many connections the function block may serve as source (SourceCount) or as sink (SinkCount). Sources and sinks are numbered in ascending order starting from 1. There can be no gaps between different source/sink numbers. A request is sent:

  ```
  Controller -> Slave: FBlockID.InstID.SyncDataInfo.Get
  ```

The answer contains the number of sources/sinks in this function block:

```
Controller -> Slave: FBlockID.InstID.SyncDataInfo.Status (SourceCount,
                                                           SinkCount)
```

Lists of synchronous channels must always be in ascending order. If not, then error code 0x06 will be returned.

3.5.2.2.1 Synchronous Source

There are two approaches of connecting a source to the network, SourceConnect and Allocate. SourceConnect uses the Connection Manager to reserve resources, which thereby have total control of resource usage. Allocate is a more decentralized approach with less control of resource usage. The Connection Manager must only use one approach per function block. But devices may support both methods.

A synchronous source provides the following functions to the network:

- **SourceInfo**
  Property SourceInfo contains detailed information about the kind of synchronous 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, DataType,
                                                         [DataDescription])
  ```

  The parameter DataType describes the kind of synchronous data stream that is sent by the source. Depending on DataType, a DataDescription may follow. Information about data types can be found in Appendix B: Synchronous Data Types.

- **SourceName**
  Property SourceName holds the name of the synchronous 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**
  Some synchronous source applications require either to start or to stop transmission of stream 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 synchronous source data stream, the abstract method SourceActivity is defined:

```
Controller -> Slave: FBlockID.InstID.SourceActivity.StartResult (SourceNr,
                                                                 Activity)
```

  Through parameter Activity = [On/Off/Pause], synchronous 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.Result (SourceNr,
                                                              Activity)
```

1. **SourceConnect Approach**
   - **SourceConnect**
     A Controller can use method SourceConnect to connect a source to already reserved channels. These channels are administrated by the Connection Manager. The Connection Manager sends:

```
Controller -> Slave: FBlockID.InstID.SourceConnect.StartResult
                                          (SourceNr, ChannelList)
```

     Upon successful execution of the method, the following is reported:

```
Slave -> Controller: FBlockID.InstID.SourceConnect.Result (SourceNr,
                                                          SrcDelay)
```

   - **SourceDisConnect**
     SourceDisConnect is used to disconnect a source from the channels it occupied. Usage of this method does not deallocate the channels that were used. These channels are administrated by the Connection Manager.

```
Controller -> Slave: FBlockID.InstID.SourceDisConnect.StartResult
                                                      (SourceNr)
```

     After the method has finished, the following is reported:

```
Slave -> Controller: FBlockID.InstID.SourceDisConnect.Result
                                                      (SourceNr)
```

2. **Allocate Approach**
   - **Allocate**
     To make the source first allocate channels and then connect to them, method Allocate is used.

```
Controller -> Slave: FBlockID.InstID.Allocate.StartResult (SourceNr)
```

     On success, the channels the source now occupies and the relative delay to the timing master is reported:

```
Slave -> Controller: FBlockID.InstID.Allocate.Result (SourceNr,
                                          SrcDelay, Channels)
```

If the allocation was not successful due to a lack of enough free channels, an error is generated with the error code "Function specific" and as Error Info the SourceNr and the number of required channels. An allocation must never be done partially. Unless all channels can be allocated, no allocation is done. The resulting error will be:

```
Slave -> Controller: FBlockID.InstID.Allocate.Error
                      ("Function Specific", SourceNr, RequiredChannels)
```

- **DeAllocate**
  DeAllocate is used by a controller wishing to cancel that which was done by Allocate before. This means that the allocated channels will be deallocated and that the source no longer is connected to them.

```
Controller -> Slave: FBlockID.InstID.DeAllocate.StartResult (SourceNr)
```

On success, the channels are no longer occupied and the source is disconnected from the channels.

```
Slave -> Controller: FBlockID.InstID.DeAllocate.Result (SourceNr)
```

3.5.2.2.2 Synchronous Sink

A function block that is used as a sink for synchronous data supports functions similar to those for a source. Error handling is also done in an analogous way.

A synchronous sink provides the following functions to the network:

- **SinkInfo**
  Property SinkInfo contains detailed information about the kind of synchronous 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, DataType,
                                                       [DataDescription])
```

The parameter DataType describes the kind of synchronous data stream that may be received by the sink. Depending on DataType, a DataDescription may follow. Information about data types can be found in Appendix B: Synchronous Data Types.

- **SinkName**
  Property SinkName holds the name of the synchronous 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 specified channels.

```
Controller -> Slave: FBlockID.InstID.Connect.StartResult (SinkNr, SrcDelay,
                                                          Channels)
```

SrcDelay is the relative delay to the timing master. It is used to provide the possibility of delay compensation. The sink returns as result:

```
Slave -> Controller: FBlockID.InstID.Connect.Result (SinkNr)
```

- **DisConnect**
  Method DisConnect is used to disconnect a sink from channels it is currently using.

  ```
  Controller -> Slave: FBlockID.InstID.DisConnect.StartResult (SinkNr)
  ```

  After the method has finished, the following is reported:

  ```
  Slave -> Controller: FBlockID.InstID.DisConnect.Result (SinkNr)
  ```
- **Mute**
  The output of synchronous data from a sink can stopped by method Mute.

  ```
  Controller -> Slave: FBlockID.InstID.Mute.SetGet (SinkNr, Status)
  ```

  Status is On or Off to turn mute on or off.

3.5.2.2.3 Handling of Double Commands

Normally a repeated synchronous control command (this means: allocate / deallocate / connect / Disconnect / SourceConnect / SourceDisConnect) should not appear. 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**
  1. **SourceConnect Utilization**
     - **SourceConnect**
       If there is a SourceConnect.StartResult command with a source number of a currently connected source and the ChannelList contains the same channels that it is connected to, a normal result message will be sent. If an already connected source is to be connected to different channels the device will first disconnect the old connection and then make the new one. Following this it sends out a result message.
     - **SourceDisConnect**
       If there is a SourceDisConnect.StartResult command, with a source number of a currently not connected source, a normal result message with the disconnected source number is sent back to the caller.
  2. **Allocate Utilization**
     - **Allocate**
       If there is an Allocate.StartResult command with a source number of a currently allocated source, a normal result message with the already allocated channels is sent back to the caller.
     - **DeAllocate**
       If there is a DeAllocate.StartResult 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.StartResult command with a sink number of a currently connected sink and the same channels that it is connected to, a normal result message will be sent. If an already connected sink is to be connected to different channels the device will first disconnect the old connection and then make the new one. Following this it sends out a result message. If a new value of SrcDelay is passed to the sink, this must be used instead of the old one.
  - **DisConnect**
    If there is a DisConnect.StartResult 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.

3.5.2.3 Compensating Network Delay

Every active node in the ring (source data bypass inactive) generates 2 samples delay for the source data (caused by internal processing). Especially in top HIFI applications, this is an unpleasant effect. The MOST system therefore provides mechanisms that allow compensation for this delay. Every MOST Network Interface Controller is provided with the information about the general delay of the entire system $\Delta T_{Network}$, and the delay up to its own node ΔT_{Node} with respect to the Timing Master. In addition to that, the delay of the active source device ΔT_{Source} must be made available by control messages.

Based on that information, the delay ΔT_{comp}, which must be compensated for, can be calculated with the help of the formula below:

$$\Delta T_{comp} = \Delta T_{Node} - \Delta T_{Source} - 2[\text{Samples}] \qquad \text{for } \Delta T_{Source} < \Delta T_{Node}$$
$$\Delta T_{comp} = \Delta T_{Network} - \Delta T_{Source} + \Delta T_{Node} - 2[\text{Samples}] \qquad \text{for } \Delta T_{Source} > \Delta T_{Node}$$

($\Delta T_{xxx} \equiv$ Contents of the respective register in the chip * 2 Samples Delay)

3.6 Handling Asynchronous (Packet) Data

3.6.1 Direct Access to the MOST Network Interface Controller

A data packet that can be sent in the asynchronous area consists of 48 Bytes of data and a 16-bit receiver address (logical node address only):

Data area MOST Network Interface Controller = 48 Byte

16 Bit										
Rx/TxLog										

The data field is significantly longer than that of the control channel. Unlike the control channel, no ACK/NAK mechanism or low level retries are implemented, since they are not necessary for most of the applications. Nevertheless, the telegram is checked. A transport protection must be implemented on a higher level.

3.6.1.1 Priorities

In every node (MOST Network Interface Controller) the priority for packet data transfer can be assigned (range 0x01, …, 0x0F with value 0x07 = lowest priority).

3.6.2 MOST Network Service

The Network Service have a mechanism, called the Asynchronous Data Transmission Service, by which the data packets described above can be sent and received. It handles the respective registers in the MOST Network Interface Controller.

3.6.2.1 Securing Data

For stream data of high bandwidth, e.g., graphical applications, it is not useful to implement mechanisms for secure data transmission. On one hand the data security (bit error rate) of MOST is approximately 10^{-12}, on the other hand, every securing mechanism must be checked by a µController, which becomes more and more difficult, as the bandwidth is increased. In case of errors, transmissions would be repeated. This would cause delays that may be unwanted. It must be decided for each application, whether mechanisms for secure data transmission would be useful, and if so, which implementation to use.

For certain applications, which transmit in the asynchronous area at a lower bandwidth, it may be useful to implement securing mechanisms. The telegram structure, quite alike that of the control channel could be used by setting TelID to 4 bits and TelLen to 12 bits.

Data Area MOST Network Interface Controller = 48 Byte (Data Link Layer 48 Bytes Mode)

16 Bit	8 Bit	8 Bit	12 Bit	4 Bit	4 Bit	12 Bit	8 Bit	8 Bit		8 Bit
DeviceID	FBlock ID	Inst. ID	Fkt ID	OP Type	Tel ID	Tel Len	Data 0	Data 1	...	Data 41

Data Area MOST Network Interface Controller = 1014 Byte (Alternative Data Link Layer)

16 Bit	8 Bit	8 Bit	12 Bit	4 Bit	4 Bit	12 Bit	8 Bit	8 Bit		8 Bit
DeviceID	FBlock ID	Inst. ID	Fkt ID	OP Type	Tel ID	Tel Len	Data 0	Data 1	...	Data 1007

TelID: Identification of kind of telegram

Meaning	TelID
MOST High Protocol User data	8
MOST High Protocol Control data	9
Reserved for MAMAC PacketsEthernet frames	A, B

TelLen: = up to 1008 (42)
Specifies the length of the data field, i.e., the number of Bytes after TelLen

Data X: Data Bytes

For securing data, MOST High Protocol is used here.

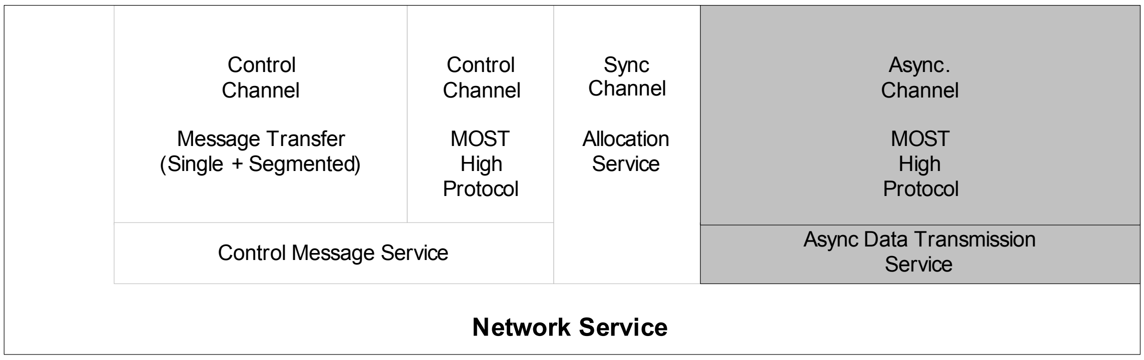


Figure 3-24: Network Service: Services for the asynchronous channel

3.6.3 MOST Asynchronous Medium Access Control (MAMAC)

To be able to run commonly used network protocols like TCP/IP (including IPX, NetBEUI and ARP) through the Asynchronous (Packet Data) channel of MOST, MOST Asynchronous Medium Access Control (MAMAC) was defined. MAMAC can be used simultaneously with the Most High Protocol.

3.7 Controlling Synchronous / Asynchronous Bandwidth

When administrating the boundary between synchronous and asynchronous data area, two contrary requirements must be taken into consideration. On one hand, there should be as much bandwidth as possible for asynchronous data transfer, so it is not reserved for unused synchronous channels. On the other hand, the boundary should be changed only in rare cases, since all synchronous connections must be re-allocated after the boundary was changed.

Even with the most adverse usage of a fully equipped vehicle, the entire available bandwidth for synchronous transfer is seldom used. Generally, less bandwidth is required for synchronous transfer.

System initialization adjusts the boundary to the center of the bandwidth. During runtime it is shifted only “to the right”, that is, in the direction of an extension of the synchronous area, up to a limit, which will reserve, e.g., one quadlet for asynchronous data transfer. There is no shifting to the left. The boundary is returned to its default value only after a re-initialization of the system.

The changing of the boundary between synchronous and asynchronous area is done physically by the Timing Master, located in the system Master device.

3.8 Connections

3.8.1 Synchronous Connections

3.8.1.1 ConnectionMaster

Synchronous connections are managed by a Connection Manager. All requests for establishing connections must be directed to this Connection Manager. It could be implemented in any device.
The Connection Manager shall supply functions defined in FBlock ConnectionMaster (0x03).

For building a point-to-point connection, FBlock ConnectionMaster provides a method **BuildSyncConnection**:

```
Controller -> CManager: ConnectionMaster.1.BuildSyncConnection.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.BuildSyncConnection.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.BuildSyncConnection.ErrorAck
                               (SenderHandle,"ProcessingError", Source, Sink)
```

Removing a connection is done in an analogous way, by using the method **RemoveSyncConnection.**

The ConnectionMaster generates an array of all existing connections including sources and sinks, where it adds more information. This array is accessible in function **SyncConnectionTable**:

```
Controller -> CManager: ConnectionMaster.1.SyncConnectionTable.Get

CManager -> Controller: ConnectionMaster.1.SyncConnectionTable.Status
                   (Source, Sink, SrcDelay, NoChannels, ChannelList,
                    Source, Sink, SrcDelay, NoChannels, ChannelList
                    Source, Sink, SrcDelay, NoChannels, ChannelList, ...)
```

The parameters are the same as those described above. SyncConnectionTable cannot be set directly. Building and removing connections is done only with methods BuildSyncConnection and RemoveSyncConnection.

After switching off the network, the contents of SyncConnectionTable are deleted, leaving no synchronous connections in the system. They must be rebuilt by new requests of the initiator(s).

The SyncConnectionTable is deleted also when Configuration.Status(NotOK) is received by the Connection Manager since all connections are removed in this case. See section 3.3.2 for more information.

FUNCTIONS				
FktID	**OPType**	**Sender**	**Receiver**	**Explanation**
BuildSyncConnection	StartResultAck	Controller	Connection Manager	Request for building connection
	ResultAck	Connection Manager	Controller	Answer with result
SyncConnectionTable	Get	Controller	Connection Manager	Request of that property, where the Connection Manager stores all active point-to-point connections
	Status	Connection Manager	Controller	Answer
RemoveSyncConnection	StartResultAck	Controller	Connection Manager	Request for removing connection
	ResultAck	Connection Manager	Controller	Answer with result

Table 3-15: Functions in ConnectionMaster in conjunction with the administration of synchronous 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 Function Block. 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.

3.8.1.2 Establishing Synchronous Connections

When building a synchronous connection, the Connection Manager uses method Allocate or SourceConnect 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 same channels as used by the source. This mechanism is explained in the following figure, and the text below.

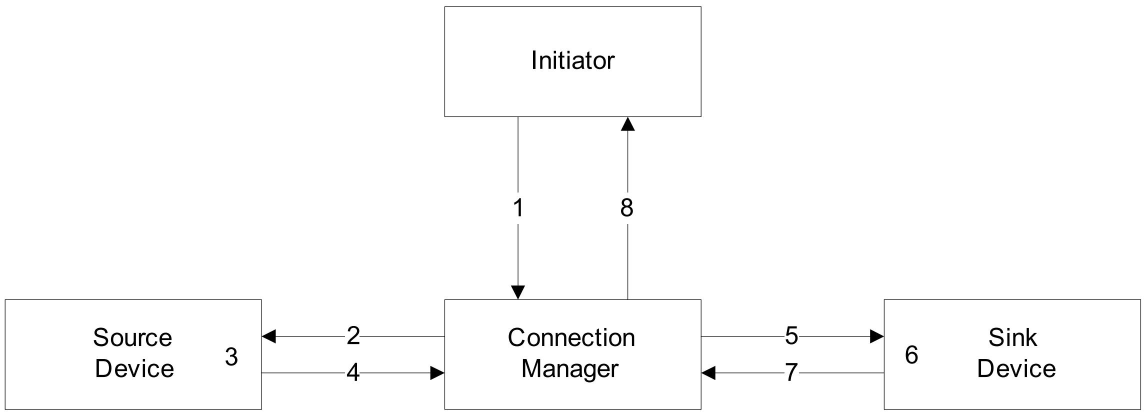


Figure 3-25: Building a synchronous connection step by step

Explanation of Figure 3-25**:**

1) Method *BuildSyncConnection* is started by an initiator, for building a connection between a source and a sink. If the source uses the method *SourceConnect* to connect to the network, the Connection Manager is responsible for the channels being unoccupied. If the method *Allocate* is used by the source the source must allocate its own channels.

2) Connection Manager sends a command to the source device to connect its synchronous output to network channels. The source must support at least one of the methods *Source Connect* or *Allocate*.

3) The Source handles the request differently depending on if it uses method *Allocate* or Source*Connect*

   a) *Allocate:*
   The source tries to allocate channels. The following results are possible:

   - Enough free channels. Reply to Connection Manager (4) as *Allocate.Result* with parameters SourceNr, SrcDelay and Channels.
   - Timing Master is busy on processing other allocation/deallocation requests. Retries may be tried until $t_{CM\_DeadlockPrev}$ has expired, at which time Allocate is regarded as failed by the Connection Manager. The rate at which the Timing Master can be asked is regulated by $t_{ResourceRetry}$.

- Not enough channels. Reply to ConnectionMaster (4) as *Allocate.Error* with parameters SourceNr and RequiredChannels.

b) *SourceConnect:*
A SourceConnect node does not need to allocate channels. It connects to the ones supplied by the Connection Manager when invoking the SourceConnect method. The result is sent back to the Connection Manager (4).

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 Channels and SrcDelay. The sink has then all the information needed of the source.

6) The Sink connects to the channels.

7) The result is sent to the Connection Manager as *Connect.Result*.

8) The ConnectionMaster reports the result of establishing the connection to the initiator by using *BuildSyncConnection.ResultAck*. If the building of the connection was successful, the ConnectionMaster 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, *BuildSyncConnection.ErrorAck* is sent and all changes to the network is unmade. That means that if an error occurred in the *Connect* process, the source is disconnected and the channels are freed again.

3.8.1.3 Removing Synchronous Connections

The initiator terminates a connection previously built by the Connection Manager. The Connection Manager commands the sink to disconnect from the channels, which it is currently using.
After a positive answer, and if the channels are not in use by another sink, the Connection Manager disconnects the source from its channels. Depending on whether the source uses SourceConnect or Allocate to connect to the network, the Connection Manager uses method SourceDisConnect or Deallocate.
After that, the Connection Manger reports the result of the termination to the initiator.

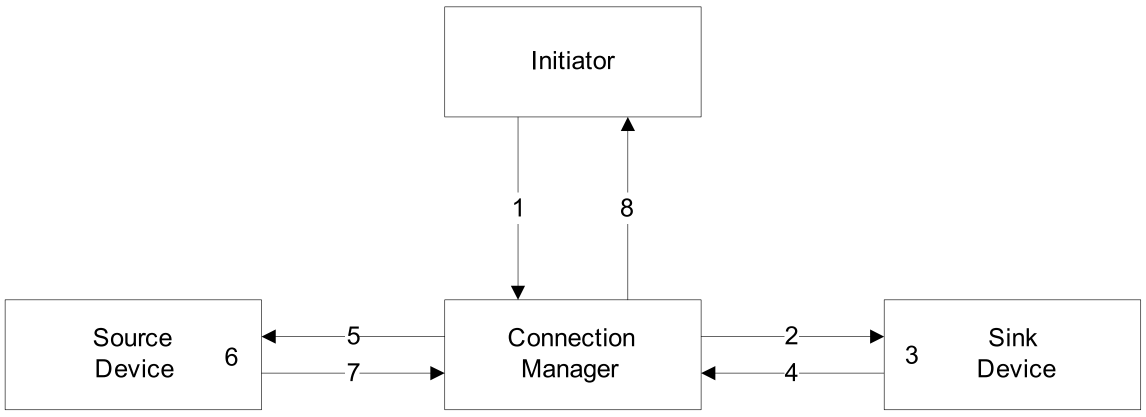


Figure 3-26: Step by step removal of a synchronous connection

Explanation of Figure 3-26**:**

1) Method *RemoveSyncConnection* 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 channels.
3) The sink disconnects from the used channels.
4) The sink reports the result to the Connection Manager by *DisConnect.Result.*
5) If no other sink uses the channels in the connection, the channels will be freed. Otherwise continue at 8. Freeing the channels is handled differently depending on if the source uses Allocate or SourceConnect. The respective method for releasing the channels is called by the Connection Manager.

6) The source disconnects or deallocates and disconnects upon reception of *SourceDisConnect* or *DeAllocate*.

a) *DeAllocate:*

The source has to deallocate the used channels. Upon trying to do this, the following results are possible:

- Successful de-allocation. Positive answer by DeAllocate.Result (7).
- The Timing Master is busy handling other allocation/ de-allocation requests. Retries may be tried until $t_{CM\_DeadlockPrev}$ has expired, at which time DeAllocate is regarded as failed by the Connection Manager. The rate at which the Timing Master can be asked is regulated by $t_{ResourceRetry}$.

b) *SourceDisConnect:*
A SourceConnect node does not need to deallocate channels. The connection manager is responsible for handling the channels. The source node disconnects from the channels and reports the result to the connection manager (7).

7) The result is sent to the Connection Manager as *SourceDisConnect.Result* or *DeAllocate.Result* depending on the method used.

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, *RemoveSyncConnection.ResultAck,* to the Initiator. The message contains the status of the requested termination of the connection.

3.8.1.4 Supervising Synchronous Connections

Every synchronous sink is responsible of supervising the validity of its output. If a source malfunctions and the data on the channels is rendered invalid the sink has to secure its synchronous output signals.

3.8.1.4.1 Enabling Synchronous Output

A sink that cannot detect the validity of the data on the channels has to verify if a device is currently putting out data. A sink application must use RemoteGetSource if it cannot make sure that it receives valid data by other mechanisms

3.8.1.4.2 Source Drops

A source that malfunctions might drop from the network, thus generating a Network Change Event. When this is detected by a sink it has to verify that a source is still connected to the channels and if not secure its synchronous output. Sinks with automatic noise detection do not need to use this method.

Another approach may be used if part of the device is still operational. The source device must then route zeros (signal mute) to the channels of the malfunctioning FBlock for a time $t_{CleanChannels}$. In this way it will be as if the sink muted its output. After time $t_{CleanChannels}$ the source must be disconnected from the network. Then the device must send out an FblockIDs.Status without the malfunctioning source FblockID. This informs the network that the source FBlock is no longer available.

3.9 Timing Definitions

T = Timer. If expired, the implementation has to invoke an error handling as defined in the respective section of the specification.

C = Timing constraint. The implementation has to fulfill this timing requirement in order to be compliant to this specification.

Name	Min Value	Typ Value	Max Value	Unit	Type	Definition
Initialization						
t_{Config}	1975	2000	2015	ms	T	Time that may pass after initialization of the MOST Network Interface Controller in a master or a slave device until a stable lock has been achieved at least once and the boundary is set to a value > 5 (i.e., the TimingMaster generated a Net On event)
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. $(63 \bullet t_{WakeUp}) + t_{WaitNodes} + t_{Lock} + t_{Boundary} < t_{Config}$
$t_{WaitNodes}$	–	–	100	ms	C	Time that may pass between start of activity at the Rx input of a device and the deactivation of its all-bypass. This timer is valid only when starting up the network.
$t_{Boundary}$	–	–	20	ms	C	Time after which a change of the boundary must be detected while waiting for the Net On event.
$t_{WaitBeforeScan}$	100	100	500	ms	T	Time between broadcast of Configuration.Status(NotOk) or NetOn and start of a new scan by the NetworkMaster. $t_{WaitBeforeScan} \leq t_{WaitAfterNCE}$
$t_{DelayCfgRequest1}$	500	500	550	ms	T	Time after which the NetworkMaster starts to query nodes again that did not answer within $t_{WaitForAnswer}$. This time is used for the first 20 attempts after the Net On event.
$t_{DelayCfgRequest2}$	10	10	11	s	T	Same as $t_{DelayCfgRequest1}$, but from the 21st attempt on.

Shutdown						
$t_{ShutDown}$	–	–	15	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 wakeup mode) and the stop of activity at the device's Tx output.
$t_{Suspend}$	1975	2000	2100	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 switching off the Tx output.
$t_{RetryShutDown}$	9.9	10	10.1	s	T	Time the PowerMaster waits between ShutDown.Start(Query) broadcasts.
$t_{Restart}$	275	300	350	ms	T	Time after switching off the Tx output until the device is ready to switch on the Tx output again. Note: This timing applies to networks with up to 19 nodes. For networks with more nodes, the following formulae applies: 1. $t_{RestartMin}$ = (Number of nodes) • $t_{ShutDown}$ – 10ms 2. $t_{RestartMax}$ = (Number of nodes) • $t_{ShutDown}$ + 65ms 3. $t_{RestartMin} \leq t_{RestartTyp} \leq t_{RestartMax}$
$t_{PwrSwitchOffDelay}$	5	5	device specific	s	T	Time between switching off the Tx output and changing to state DevicePowerOff
$t_{SlaveShutdown}$	16	–	–	s	T	Time a slave device shall wait after ShutDown.Start(Execute) before it may switch off light without detection of no light at the input.

General						
t_{Lock}	75	100	115	ms	T	Time during which no lock errors must occur, before the lock is declared “stable”. Note: While a ring break diagnosis is pending, t_{Diag_Lock} has to be used instead.
t_{Unlock}	60	70	100	ms	T	Accumulated time of unlocks that lead to the detection of a critical unlock.
$t_{MPRdelay}$	–	–	200	ms	C	Time between a network change event (NCE) and the notification of applications for which NCEs are relevant.[1] $t_{MPRdelay} \geq t_{Lock}$
$t_{WaitAfterNCE}$	200	200	500	ms	T	Time between a NCE and the start of the network re-scan by the NetworkMaster. $t_{MPRdelay} \leq t_{WaitAfterNCE} \leq t_{DelayCfgRequest1}$
t_{Bypass}	50	70	100	ms	T	Time the all-bypass of a device must stay active after being activated. This timer is not valid when starting up the network. It applies only when a node drops out of the network. $t_{Bypass} \leq t_{WaitNodes}$
t_{Answer}	–	–	50	ms	C	Time during which a network slave must respond to a query by the NetworkMaster.
$t_{WaitForAnswer}$	100	200	500	ms	T	Time a NetworkMaster waits for an answer from a queried slave. $t_{WaitForAnswer} \leq t_{DelayCfgRequest1}$
$t_{ResourceRetry}$	–	–	10	ms	C	Time between attempts of allocating or deallocating synchronous resources
$t_{Property}$	–	–	200	ms	C	Time between complete reception of a query to a property and the start of the response message.
$t_{WaitForProperty}$	250	300	350	ms	T	Time a shadow waits for the reception of a query to a property.
$t_{ProcessingDefault1}$	–	100	150	ms	T	Time a device waits before sending the first Processing message.
$t_{ProcessingDefault2}$	100	100	–	ms	T	Time a device waits between sending subsequent Processing messages.

[1] Devices containing such applications must only be used in node positions where this requirement can be fulfilled.

$t_{WaitForProcessing1}$	200	200	250	ms	T	Time a Shadow waits for the reception of the first Processing message, if not specified otherwise for the respective message in the FBlock Specification.
$t_{WaitForProcessing2}$	200	200	-	ms	T	Time a Shadow waits for the reception of the following Processing messages. The timer should be set to 100 ms more than $t_{ProcessingDefault2}$.
$t_{CleanChannels}$	3.5	5	25	ms	T	Time during which a source must route zeroes onto synchronous channels before it stops using them.
$t_{CM\_DeadlockPrev}$	-	-	1000	ms	T	Timer to prevent deadlocks in the connection building process.
Ring Break Diagnosis Note: The following definitions represent the range within system integrators can choose the timings for a specific system. Tolerances that are applied to these values for nodes of the same system are: +15ms, -25ms.						
t_{Diag_Light}	0	0	100	ms	T	Time a node waits for activity at its Rx input before switching to ring break master mode.
t_{Diag_Master}	2	28	58	s	T	Time a node stays in ring break master mode before generating the result of the diagnosis.
t_{Diag_Slave}	4	30	60	s	T	Time a node 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_Lock}	250	250	260	ms	T	Same as t_{Lock}, but valid during ring break diagnosis.
t_{Diag_Start}	0	0	10	s	C	If using SwithToPower to trigger ring break diagnosis, the diagnosis has to be started within t_{Diag_Start}.
$t_{Diag\_Restart}$	0	0	10	s	T	This is the time a device has to wait after an unsuccessful diagnosis (ring broken) until it can be restarted by network activity. The value of this timer should be greater or equal to the maximum difference between the startup of ring break diagnosis in different devices. If this time is unknown, the maximum value can be used.

Table 3-16: Timing Definitions

3.10 Secondary Node

For some applications it can be useful to integrate two MOST Network Interface Controllers into one device. This section describes, which scenarios are available, and how the tasks are divided up between the two MOST Network Interface Controllers.

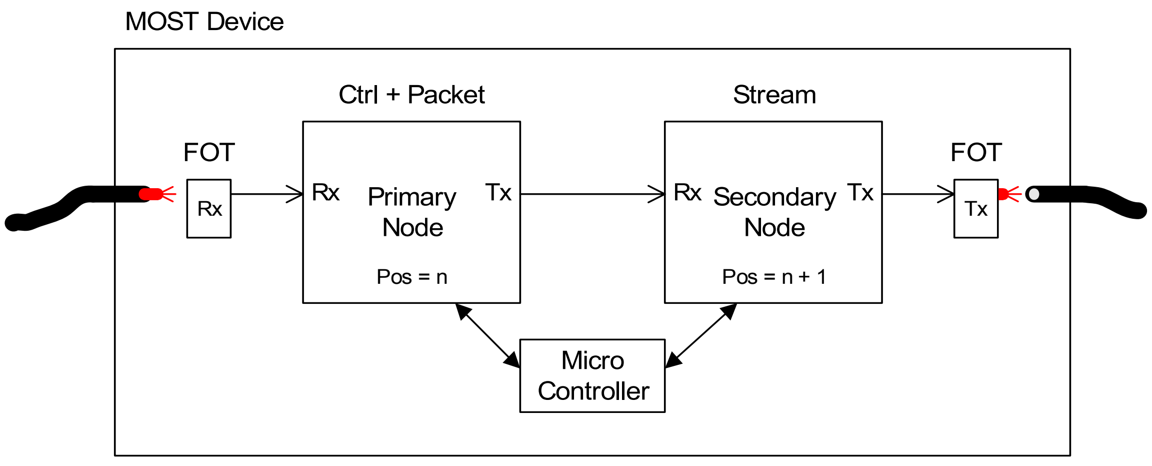
Please note:
The Secondary Node approach applies for Timing Slave nodes only.

When using secondary nodes, both MOST Network Interface Controllers are controlled by the same microcontroller. That node, where Control Messaging is handled (and where the Network Service are running mainly) is called "Primary Node". There are three scenarios:

- Scenario 1:
  Primary node : Ctrl + Packet
  Secondary node : Stream
- Scenario 2:
  Secondary node : Stream
  Primary node : Ctrl + Packet
- Scenario 3:
  Primary node : Ctrl + Stream
  Secondary node : Packet

All timing constraints, which apply for "normal" MOST nodes must be fulfilled by secondary nodes too. The address of the secondary node should be determined and initialized too.

3.10.1 Scenario 1

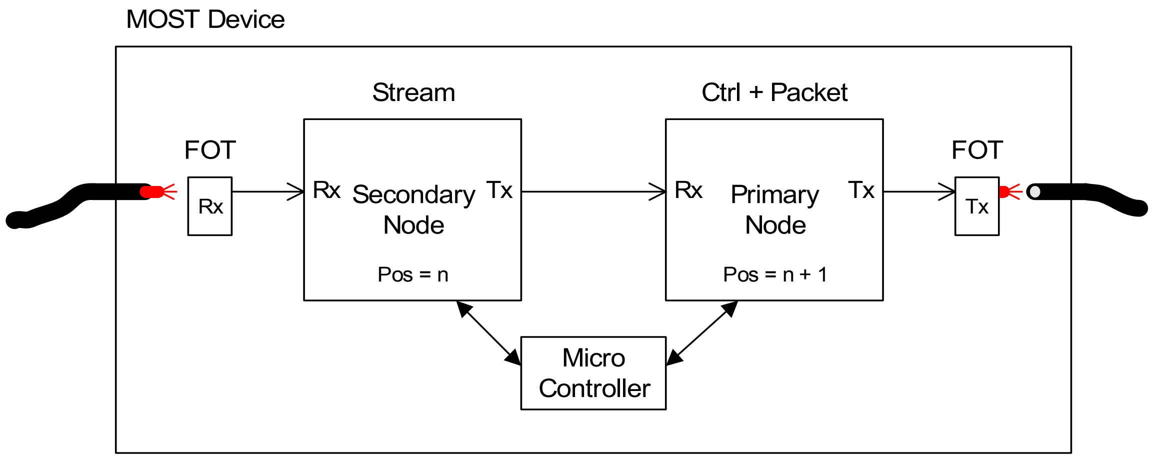


Primary Node		Secondary Node	
- SP Parallel Async.,	CP Serial	- SP Serial,	CP Serial
- SP Parallel Async.,	CP Parallel	- SP Parallel Sync.,	CP Serial
		- SP Parallel Sync.,	CP Parallel

Figure 3-27: Secondary Node, scenario 1

In Scenario 1, the node position (Pos) of the primary node is always less than the node position of the secondary node. Figure 3-27 shows, which configurations for Source Data Port (SP) and Control Port (CP) are available.

3.10.2 Scenario 2

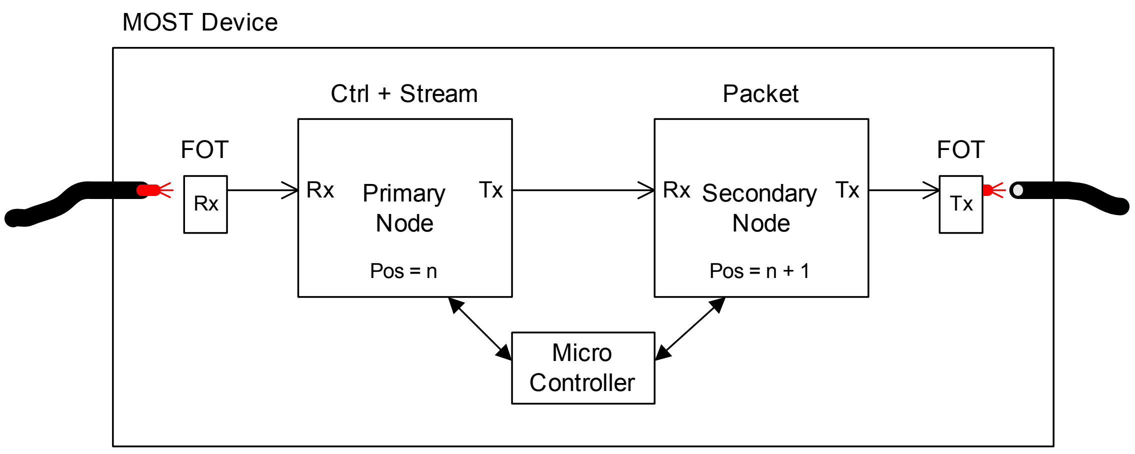


Secondary Node		Primary Node	
- SP Serial,	CP Serial	- SP Parallel Async.,	CP Serial
- SP Parallel Sync.,	CP Serial	- SP Parallel Async.,	CP Parallel
- SP Parallel Sync.,	CP Parallel		

Figure 3-28: Secondary Node, scenario 2

In Scenario 2, the node position (Pos) of the secondary node is always less than the node position of the primary node, except for the primary node being the system's Timing Master. Figure 3-28 shows, which configurations for Source Data Port (SP) and Control Port (CP) are available.

3.10.3 Scenario 3



Primary Node		Secondary Node	
- SP Parallel Sync.,	CP Serial	- SP Parallel Async.,	CP Serial
- SP Parallel Sync.,	CP Parallel	- SP Parallel Async.,	CP Parallel

Figure 3-29: Secondary Node, scenario 3

In Scenario 3, the node position (Pos) of the primary node is always less than the node position of the secondary node. Figure 3-29 shows, which configurations for Source Data Port (SP) and Control Port (CP) are available.

4 Hardware Section

4.1 Basic HW Concept

The fundamental hardware structure of a MOST device is displayed in the block diagram below. There are blocks that are not mandatory, since, e.g., a simple MOST device does not always need a micro controller (active speaker). Areas that are not mandatory are displayed in gray. A MOST device consists of:

- Optical interface area
- MOST function area
- µC area
- Application area
- Power supply area

A MOST network is awakened optically. All MOST devices are connected to a continuous power supply. They activate a sleep mode if required. If a device is in sleep mode, the power consumption should be reduced as far as possible (current < 100 µA). For this reason, unused areas must be separated from the power supply. Only the sections that are absolutely necessary will stay powered. It must be taken into consideration that no parasitic currents will flow via signal lines between inactive and active sections.

The individual areas are explained below.

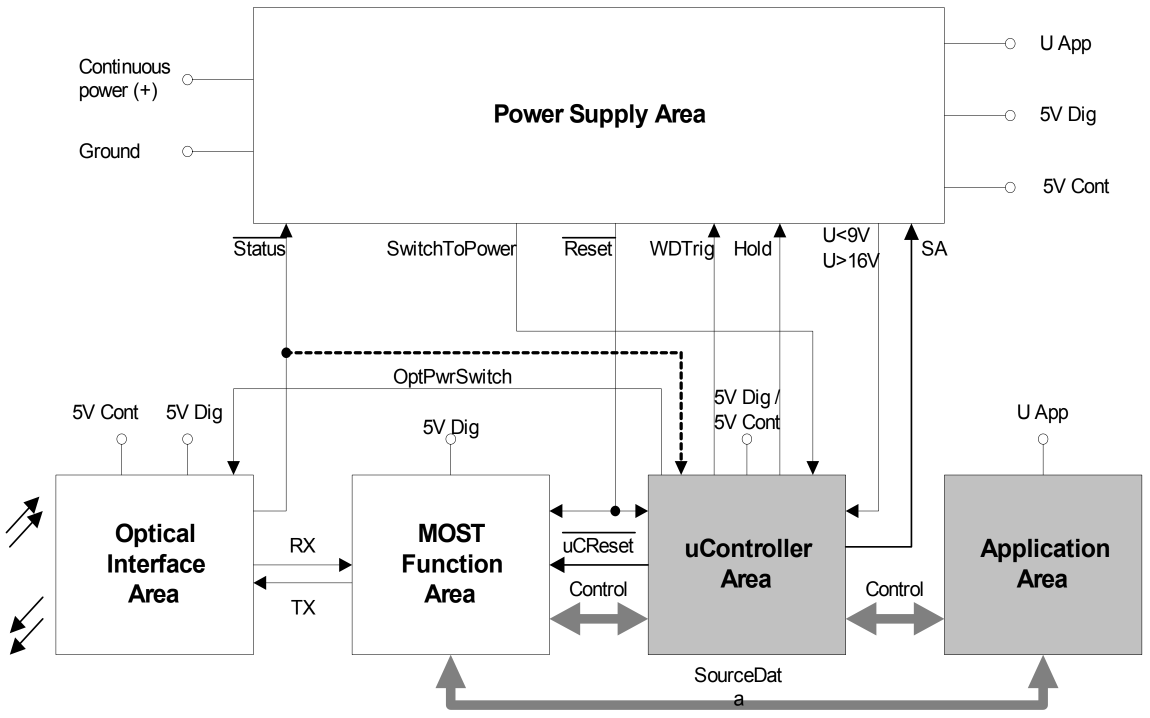


Figure 4-1: Example of the structure of a MOST device, the different functional areas and their interfaces

4.2 Optical Interface Area

4.2.1 Overview

The optical interface area consists of an optical receiver and an optical transmitter. Both communicate with the MOST Network Interface Controller via a single data line (RX and TX). The receiver is connected to continuous power, unlike the transmitter.

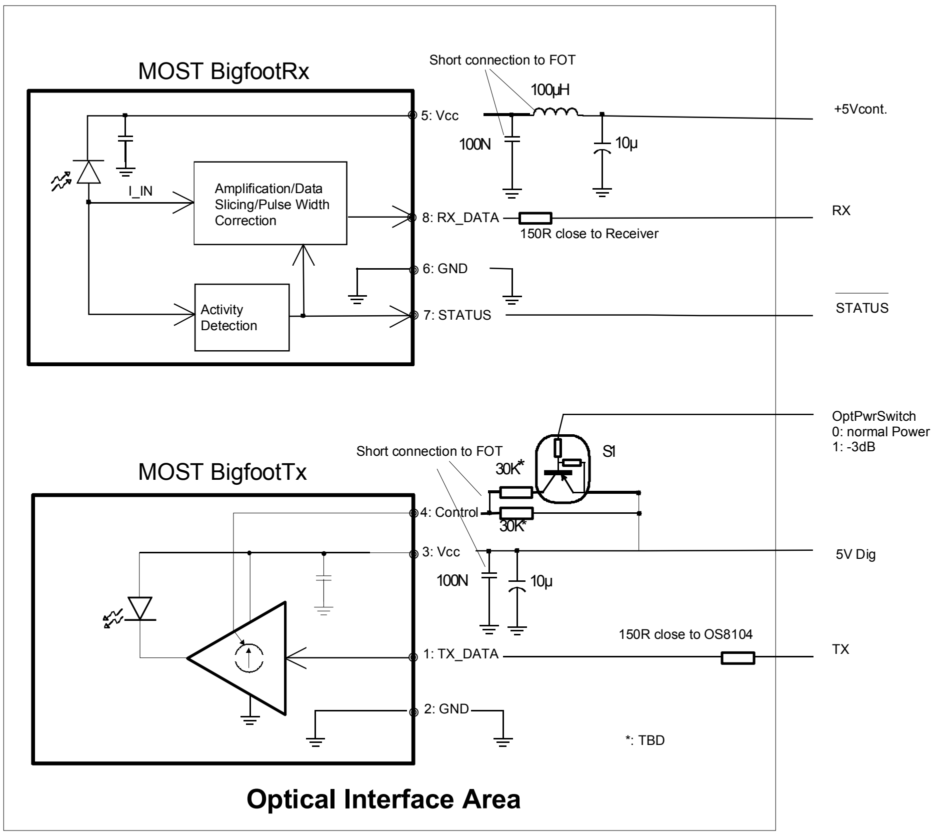


Figure 4-2: Optical interface area

The receiver contains an ActivityDetection logic that is supplied with continuous power via the micro power regulator (5V cont.), and that consumes less than 20µA. As soon as the ActivityDetection logic recognizes modulated light, signal status is switched to low, and the received data stream is switched to the output.

Signal **Status** is connected to the power supply area, and therefore the power to the MOST Network Interface Controller, and eventually to other areas, is switched on.

If no light is received, the receiver is switched off, except for the wake-up logic. Signal **Status** is high then.

It is possible to influence the driver current of the transmitter by a resistor between 5Vdig and input Control. A second resistor that has the same value can be connected in parallel to the first resistor by using switch 1. When doing that, the optical output power is increased by approx. 3dB. Switch 1 is controlled by signal **OptPwrSwitch**, which is driven by the µC area. The µC only activates **OptPwrSwitch**, if it has received the respective KWP2000 command. After $t_{OptPwrLow}$, the µC has to deactivate OptPwrSwitch independently.

In normal operation mode the switch is closed, so the transmitter runs at maximum optical power output.

This circuit provides diagnosis. If the optical power between two devices is reduced by 3dB and the system still works correctly, it can be assumed that there is a reserve of at minimum 3dB with respect to the optical power budget.

Please note:
By an appropriate arrangement (e.g., Pull down resistor, or inverter connected to OptPwrSwitch) it must be made sure, that S1 is closed in case of an inactive µC.

For the lock behavior in a MOST network there are two important influencing variables:

- Optical Power Budget
- Phase jitter

In opposite to the power budget, the phase jitter may be accumulated when passing several nodes. An important influence variable for the phase jitter is the design of the optical interface area. Here interference's on highly resistive data wiring and crosstalk may occur.

For avoiding that, in the data lines to and from the MOST Network Interface Controller a resistor of 100 Ohms up to 150 Ohms must be inserted. The resistors must be placed as close to the feeding output as possible. In addition to that, optical receiver and transmitter must be placed closely to the MOST Network Interface Controller. The maximum length of data lines to Rx and from Tx must be less than 1.5 cm.

Another important factor is the layout of the PCB. Below all data lines, transmitter and receiver a HF ground plane should be placed. Ideally, Rx and TX line are placed as far apart as possible, separated by a piece of ground area. The shielding box of the optical header must be connected well to the ground plane (by soldering). In addition to that, the power supply of the optical interface area must be buffered and blocked carefully. Therefore the bypass capacitors must be placed as close to the transmitter and receiver as possible. 100 nF (Ceramic type) must be placed between every VCC and GND. Another important point is, that the bypass capacitors of transmitter and receiver must be located between the transmitter/ receiver and that point, where the two ground planes of receiver and transmitter are connected together.

Please note
Since very high data rates are transported at low signal levels, Optical Interface Area and MOST Function Area must be designed with respect to the rules of high-frequency engineering.

4.2.2 Connection Systems (Pig Tail)

In contrast to existing systems both transceivers are placed remotely with respect to the device's plug; e.g., on the PCB near by the MOST Network Interface Controller. They are connected to the device's socket with "pig tails" (Pieces of POF). This provides the following advantages:

- Possibility of protecting the end of the fiber at the device's socket
- Easing of EMI problems
- Flexible placement on PCB
- Small dimensions
- Decoupling of the plugging system of the device from the case of the FOTs

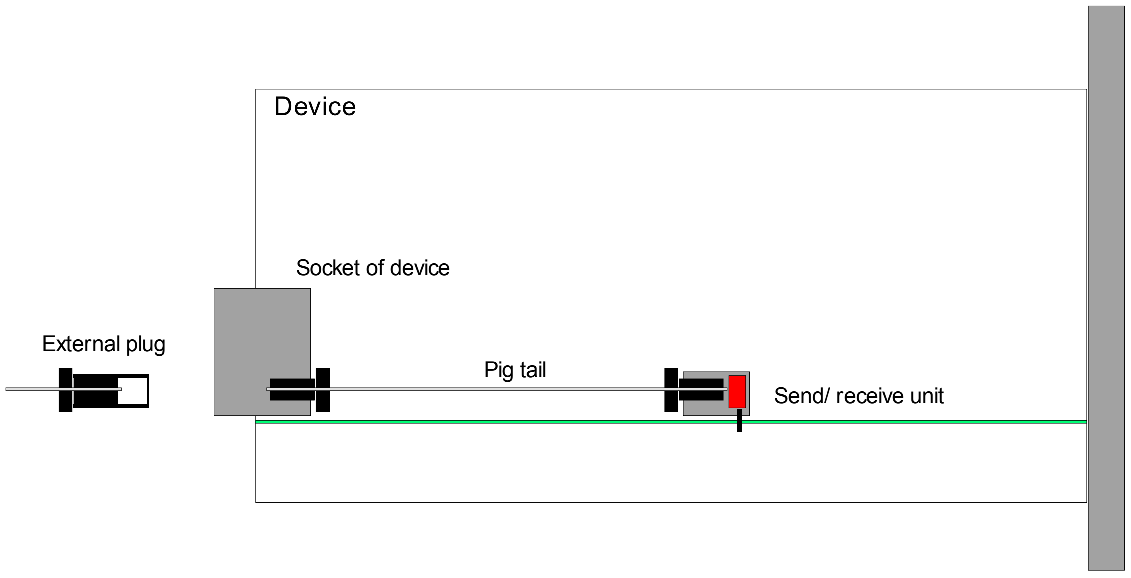


Figure 4-3: Connecting the FOTs to the plug of the device via "pig tail"

The "pig tails" are connected to the FOTs and to the device's plug with a plugging system in each case.

The device's plugging system is carried out modularly and in a hybrid way. So one or more pairs of optical connectors can be combined with different electrical connectors as in a kind of model kit, for deriving plugging systems for the different devices.

Please note:
The description above is one possible implementation of a "pig tail". Also other solutions are possible. However, the mechanical interface is standardized.

4.3 MOST Function Area

The MOST function area consists of the following components:

- MOST Network Interface Controller
- Crystal
- PLL-Filter

The MOST Network Interface Controller communicates with a microcontroller via I^2C (as slave), SPI or parallel bus. Source data is exchanged with the application via the source data bus.

Reset:
On a reset, the MOST Network Interface Controller activates all bypass mode, switches to Slave mode, and switches the interrupt pin to inactive ('1'). After reset is deactivated, the interrupt pin changes to '0' (PowerOn interrupt). The microcontroller (µC) waits for this interrupt and then initializes the MOST Network Interface Controller in Timing Master or Slave mode.

For devices with µC area it must be possible in any case, that the MOST Network Interface Controller can be reset by the respective µC as well (µC reset or Watchdog Reset), since here the MOST Network Interface Controller is not controlled and initialized via the network, but by the µC.

Please note:
During Reset (not software reset), the signal RMCK is not valid. If RMCK is used as device clock, this must be taken into consideration.

4.4 µC Area

The microcontroller (µC) area mainly consists of the µC and some memory, and is not mandatory for a MOST device. In the case of devices with a µC area, there may be applications that are tightly coupled to the network activity. They need to realize a low standby-current I_{STBY}, so in PowerOff mode of the network, the µC must be switched off.

At the same time there are devices, which must be active even if there is no network activity. Here the µC area must be connected to a continuous power supply.

In addition to that, there are devices, which are to be arranged in between. They are active without network activity, but are not connected to continuous power (for example, the power supply of a CD changer during eject of disc).

4.5 Application Area

Application area refers to the application peripherals such as receivers, amplifiers, drives, etc. The way of implementing an application area is very device-specific. In some devices, especially those with application peripherals that have high power consumption, it makes sense to supply the peripherals separately from the logic, i.e., the µC area and the MOST function area, in order to switch them on and off separately. In other applications, the application area must be connected to a continuous power supply.

If internal communication is required, the MOST Network with all devices connected to it is powered. Since this may happen also in such cases where the vehicle is parked, the power consumption in this communication mode (Logic_Only_Mode) must be kept as low as possible (not only in sleep mode). This means, that it must be possible to remove the Application Area from power (if procurable).

4.6 Power Supply Area

Please note:
The voltage levels shown in this section here could vary between the systems. Therefore, they are non-normative and not specified in detail. Binding values must be defined in the specifications of the System Integrators. The definition and relation between voltage levels can be found in section 4.7. This chapter describes the power supply for a device that is usually active when the network is active, so a low standby-current I_{STBY} must be achieved. This is the most complex case. Figure 4-4 shows an example for the implementation of a Power Supply Area.

To meet these requirements, a MOST Network Interface Controller, microcontroller (µC), and application peripherals are completely separated from power. In addition to that, the application periphery is powered separately, so that it can be switched off although the logic is still running (e.g., drive).

The implementation of the power supply area, as shown in Figure 4-4, mainly consists of:

- Filter, unload-protection, EMI/EMC protection
- Micropower regulator (5V Cont.)
- SwitchToPower detector (optional)
- Power on logic
- Digital power supply (5V Dig)
- Application power supply (U App)
- Bad power condition comparator
- Reset generator
- Watchdog timer

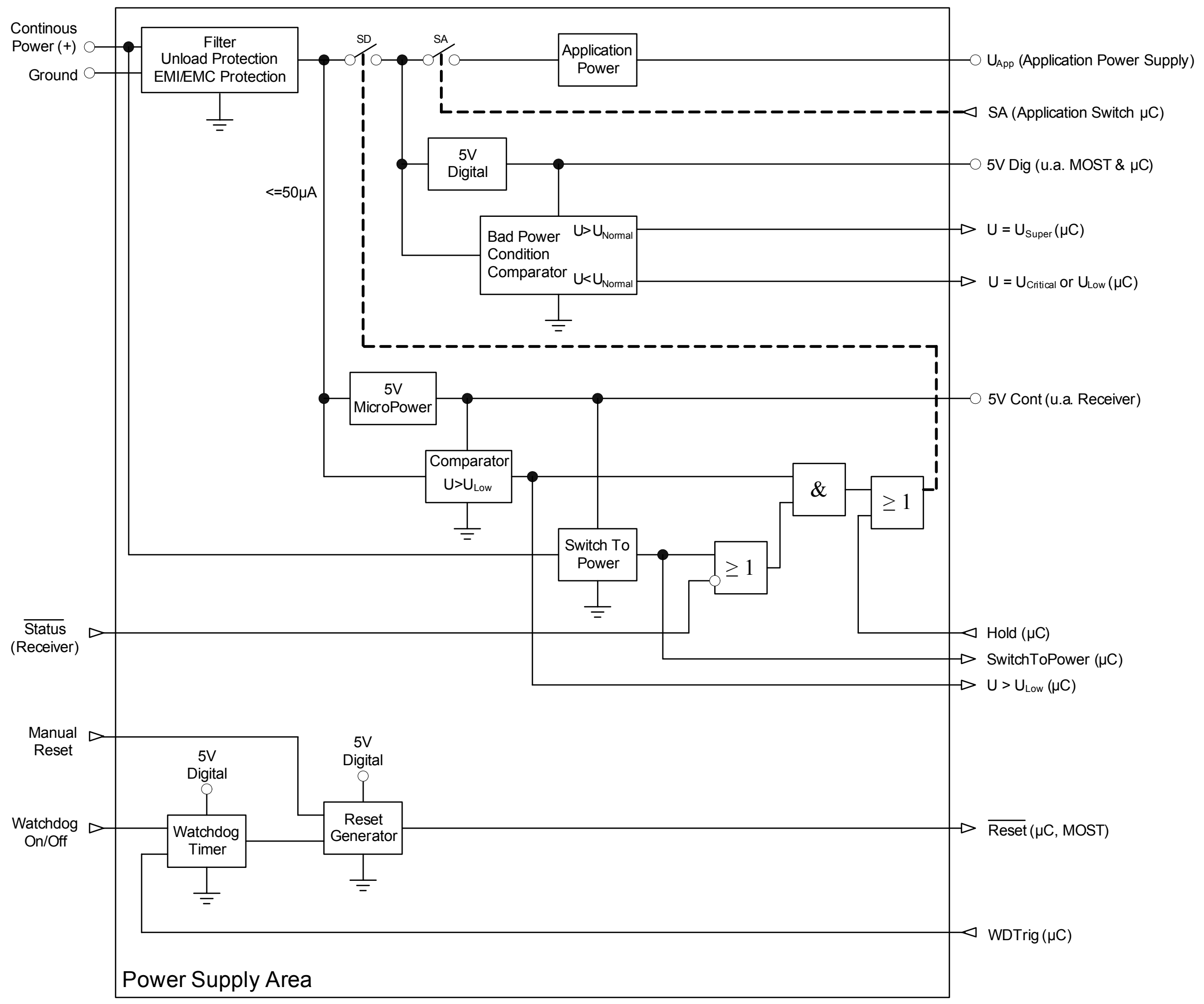


Figure 4-4: Block diagram of power supply area

Filter and **EMI/EMC-Protection** filters the power supply and protects the device from incoming radiation, or it prevents the device from sending out radiation. The **Unload-Protection** provides the overcoming of short periods of low voltage.

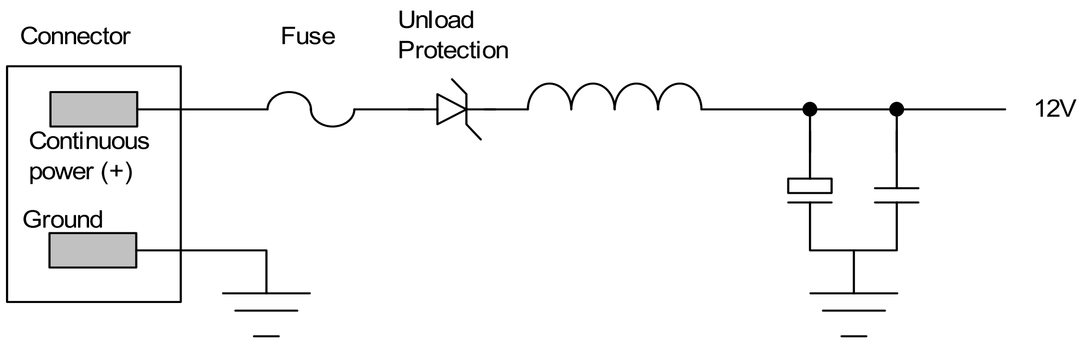


Figure 4-5: Input section of power supply area

The **Micropower Regulator** provides power supply for the receiving FOT unit with wake-up functionality and of the Power On logic, if the device is switched off on an inactive bus. Furthermore, it can be used to supply volatile memory devices. In total, the device has to meet a manufacturer-specific standby current I_{STBY}. In case of the devices that stay active on an inactive network, or that become active from time to time on an inactive network, the **Micropower Regulator** must be dimensioned to provide more power.

The **SwitchToPower Detector** is used for ring break diagnosis, where the location of an interruption of the ring is localized. This is not done during normal operation, but in the car repair, or at the assembly line. Note that the SwitchToPower Detector is optional.

Since the bus cannot work properly on a ring break, the devices must get a trigger in another way. Such a trigger is set through a defined switching off of the power supply of all devices for some seconds by a central power switch. The switched-off state should be maintained for some seconds, because all devices should be completely unloaded.

Ring break diagnosis is started by switching on power by the central power switch. The SwitchToPower detector recognizes that the device powered up, and generates a pulse, by which power of the device stays activated for a certain time. After the reset phase, the micro controller (µC) recognizes with the help of the SwitchToPower signal that the device was powered, and switches to ring break diagnosis mode. Before this, Hold must be activated to prevent the device from being switched off again.

If no communication is started on the network, the µC must deactivate Hold so the device can switch back to sleep mode.

The SwitchToPower detector must be implemented so that the SwitchToPower pulse is generated only if the power sinks below a certain threshold. Under no circumstances should short breakdowns on the supply voltage (e.g., by the starter) lead to a SwitchToPower pulse.

Therefore the SwitchToPower detector gets armed only, if the device was separated from power for at least 2 seconds, and at most 4 seconds (2 sec < t1 < 4 sec). Only then will it generate a pulse when the device is connected to power. This must be made sure of with the help of suitable measures (unload protection diode, and individual electrolyte capacitor at the power supply line of the detector).

In addition to that, the SwitchToPower detector must supervise the power supply before unload protection, since, caused by the switching off of all function areas, the voltage will decrease very slow.

The SwitchToPower pulse must have a minimum length t2, which must be long enough for the µC to safely recognize the pulse.

This is shown in the figure below for ideal signals (vertical edges). The SwitchToPower detector should be implemented at low cost, and in a way so that it works on ideal conditions as shown in the figure below. It should not be tried (at high cost) to meet the timing exactly, even on non-ideal conditions, since imprecise behavior of devices can be compensated for by the duration of the real trigger, and it is not very critical if, on an alleged trigger (e.g., caused by starting the engine), a device inadvertently switches to diagnosis mode.

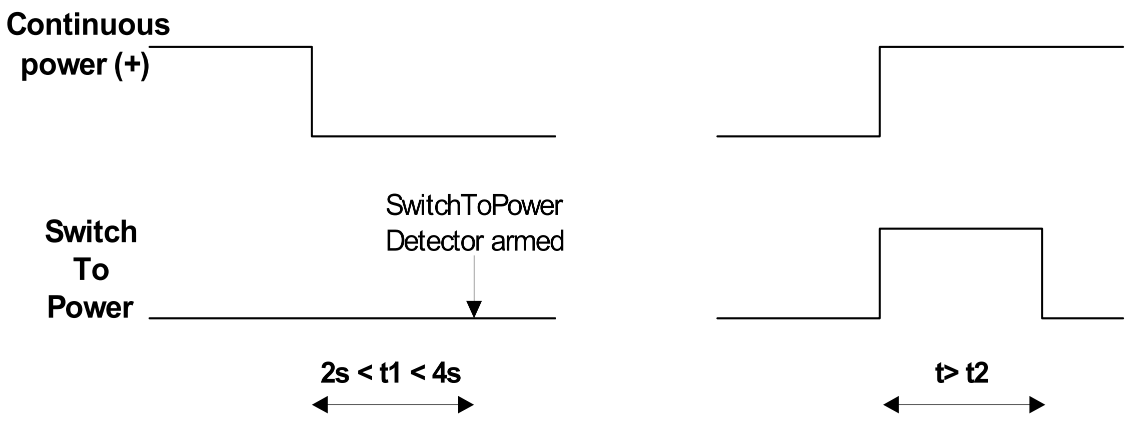


Figure 4-6: Timing of the output signal of the SwitchToPower detector depending on voltage at continuous power input

The **Power On Logic** checks to see whether the bus is active, or if the SwitchToPower detector indicates that the device is freshly connected to power. If, in addition to that, the U > U_{Low} comparator indicates a sufficient supply voltage, switch SD is closed and **Digital Power Supply** is connected to power. **Digital Power Supply** then supplies the MOST Network Interface Controller and the microcontroller (µC). As soon as the µC is started, it keeps switch SD closed by an additional input to the Power On Logic (Hold).

Later on, the µC decides if and when the application periphery (application area) will be powered, and activates SA.

The **U_{Low} comparator** indicates whether the input voltage is above the U_{Low} range or not. It is important to implement a hysteresis here, since when switching off the supply voltage due to low voltage, the voltage at the input of the comparator will suddenly be increased again. Without hysteresis, the device would be switched on again, leading to an oscillation of the U_{Low} comparator, and of the entire digital supply voltage.

Please note:
The hysteresis must be implemented in a way that the output signal of the U_{Low} comparator is switched off, when the voltage drops to U_{Low}. The output signal of the U_{Low} comparator must then be switched on again only, if the voltage rises to U_{Normal}.

The U_{Low} comparator must behave in a defined way, even if the voltage keeps decreasing and the micro power regulator does not stabilize its output voltage, but follows the input voltage only. That means that the U_{Low} comparator must prevent the device from switching on even when reaching low voltages (e.g., < 2V...3V).

The **Bad Power Condition comparator** recognizes critical voltage ($U_{Critical}$) and super voltage (U_{Super}) on the supply power, so that appropriate actions can be taken. For the Bad Power Condition signals, a hysteresis is not mandatory, since they do not control switching off power. The signals are evaluated only by the micro controller (µC).

The **Reset Generator** generates reset for the MOST Network Interface Controller and eventually for a µC if available. It is mandatory for all devices! Possible sources for reset are:

- Device connected to power
- Transition between low voltage to normal operation
- Low voltage on power
- Manual reset (reset button)
- Watchdog timer

The maximum length of the reset pulse is 300ms.

If a µC is available, a **Watchdog Timer** (eventually with an integrated reset generator) is mandatory. The watchdog timer initiates a reset at the reset generator, when not triggered by the µC for a certain time (WDTrig). This closes the all-bypass of the MOST Network Interface Controller. Even if the application processor does not restart, the device behaves in a neutral manner with respect to the bus. If a device has no µC, no watchdog timer is required.

Please note:
It must be made sure, that the HOLD mechanism (by which the µC keeps the device powered) is reset as well. The MOST Network Interface Controller can be reset by the µC as well.

4.7 Voltage Levels

In general, a device in sleep mode must not wake the bus (light on), caused by low voltage or super voltage. Four voltage ranges are defined:

Normal operation (U_{Normal}):

Device works normally, all functions are within the specified tolerances.

Super voltage (U_{Super}):

The device is in a safe operation state, which must be defined for each device individually.

Critical voltage ($U_{Critical}$):

The device is in a safe operation state, which must be defined for each device individually. The NetInterface works normally, the device can communicate. On a recovery from this state, the network does not need to be initialized again.

Low voltage (U_{Low}):

The device is in a safe operation state, which must be defined for each device individually. The voltage has dropped to a value where the device cannot communicate for long. The NetInterface does not work any longer, so a device that cannot communicate safely has to switch off light in a safe way.

The following relation holds between the different levels: $U_{Low} < U_{Critical} < U_{Normal} < U_{Super}$

A safe operation state means that the device must take measures for avoiding failure, overheating, or destruction of its own or connected functional sections. In addition, it must switch to a state from which it can resume working normally if normal voltage is restored.

Examples:

- Muting and eventually switching off of amplifiers (danger of overheating, protection of loudspeakers when switching off caused by low voltage).
- Switching off the servo units of CD/MD player (protecting the optical PickUp).

Remarks:

1. The device must be able to work between $U_{Critical}$ and U_{Super}, and the critical voltage area reaches down to U_{Low}. It could be tried e.g., to enter Low Voltage as late as possible. Especially when using switched power supplies, it can be possible to drop the Low Voltage threshold to e.g., 3 V.
2. Hysteresis ranges must be implemented for avoiding oscillating!

Low voltage for a short period of time:

Some devices need a long time for initialization (Operating system, system communication...). If such a device would be reset even at short pulses of low voltage, it needed to be initialized after that. The interruption that would occur with respect to the entire system would be recognizable by the customer (e.g., interruption of audio when starting the engine). Such a device should be able to survive short Low Voltage periods, without the need of being re-initialized. Especially the initialization status of the µC must be secured. This may be done, e.g., by using buffer capacitors, unload protection diode, a separate power supply for the digital section, releasing of the application peripherals, stopping the µC, etc. Also the operation of the NetInterface can be reduced, e.g., by resetting the MOST Network Interface Controller, which will then close its all-bypass (except a device containing the Timing Master). The light should be kept switched on as long as possible, since then the rest of the system would not be disturbed. After the Low Voltage period the MOST Network Interface Controller will be re-initialized (in total < 100ms). For more information about the behavior of the software in case of Low Voltage please refer to section 3.2.5.8 on page 149.

5 Appendix A: Network Initialization

This Appendix contains some behavioral examples regarding Network Management. Requirements regarding Network Management behavior of the Network Master and the Network Slaves can be found in section 3.2 and MOST Dynamic Specification.

5.1 Network Master Section

This section contains scenarios of the behavior of the Network Master during network initialization.

5.1.1 System Startup when a Central Registry is Available

This section contains an example of how the Network Master behaves when initializing the system with a stored Central Registry.

1. Starting Situation:

Desired configuration:

Rx/TxLog	FBlockID	InstID
0x0101	AudioDiskPlayer	1
0x0102	AM/FMTuner AudioDiskPlayer	1 2
0x0103	TVTuner	1
0x0104	AudioAmplifier AudioAmplifier	1 2

Central Registry of last system run

Rx/TxLog	FBlockID	InstID	Available?	Position
0x0101	AudioDiskPlayer	1		
0x0102	AM/FMTuner AudioDiskPlayer	1 2		
0x0103	TVTuner	1		
0x0104	AudioAmplifier AudioAmplifier	1 2		

2. System Startup - Checking Configuration: Devices 0x0102 and 0x0103 Available

Rx/TxLog	FBlockID	InstID	Available?	Position
0x0101	AudioDiskPlayer	1	No	
0x0102	AM/FMTuner AudioDiskPlayer	1 2	Yes	1
0x0103	TVTuner	1	Yes	2
0x0104	AudioAmplifier AudioAmplifier	1 2	No	

⇒ **Broadcast Configuration.Status (OK)**

3. Supplementary Registration: Device 0x0104 Joins Network (either physical through NetworkChangeEvent, or logical); checking configuration

Rx/TxLog	FBlockID	InstID	Available?	Position
0x0101	AudioDiskPlayer	1	No	
0x0102	AM/FMTuner AudioDiskPlayer	1 2	Yes	1
0x0103	TVTuner	1	Yes	2
0x0104	AudioAmplifier AudioAmplifier	1 2	Yes	3

⇒ **Broadcast Configuration.Status (Control=New, AudioAmplifier.1, AudioAmplifier.2)**

4. Shutdown: Normal Shutdown with ShutDown.Start; Store error “Device 0x0101, AudioDiskPlayer.1 missed”

3b. Variant: Un-initialized device 0xFFFF joins network at node position 2; checking configuration; recognizing un-initialized device in system; broadcast Configuration.Status (NotOK); building addresses; building central registry

Rx/TxLog	FBlockID	InstID	Available?	Position
0x0101	AudioDiskPlayer	1	No	
0x0102	AM/FMTuner AudioDiskPlayer	1 2	Yes	1
0x0103	TVTuner	1	Yes	3
0x0104	AudioAmplifier AudioAmplifier	1 2	Yes	4
0x0102	Speech Recognition	1	Yes	2

⇒ **Broadcast Configuration.Status (OK)**

5.1.2 Flow of System Initialization Process by the Network Master

The flow in Figure A-5-1 shows how the Network Master initializes the system. Refer also to Figure A-5-2 for a flow of how the Network Master performs the configuration requests from the Network Slaves during the system configuration.

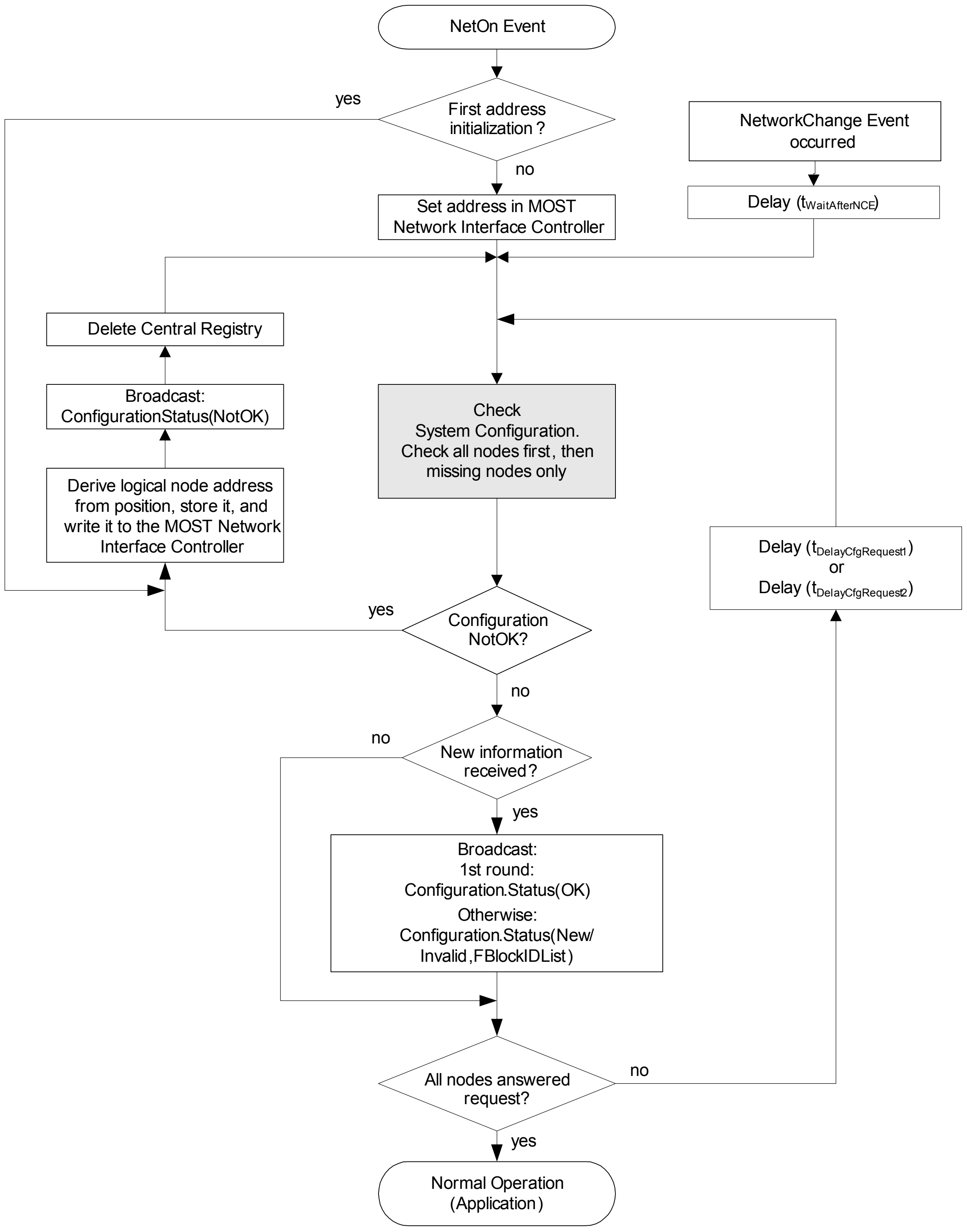

NetOn Event
yes
First address initialization ?
no
NetworkChange Event occurred
Set address in MOST Network Interface Controller
Delay ($t_{WaitAfterNCE}$)
Delete Central Registry
Broadcast: ConfigurationStatus(NotOK)
Check System Configuration. Check all nodes first, then missing nodes only
Derive logical node address from position, store it, and write it to the MOST Network Interface Controller
Delay ($t_{DelayCfgRequest1}$) or Delay ($t_{DelayCfgRequest2}$)
yes
Configuration NotOK?
no
no
New information received?
yes
Broadcast: 1st round: Configuration.Status(OK) Otherwise: Configuration.Status(New/ Invalid,FBlockIDList)
no
All nodes answered request?
yes
Normal Operation (Application)

Figure A-5-1: Flow of initialization on application level in a NetworkMaster

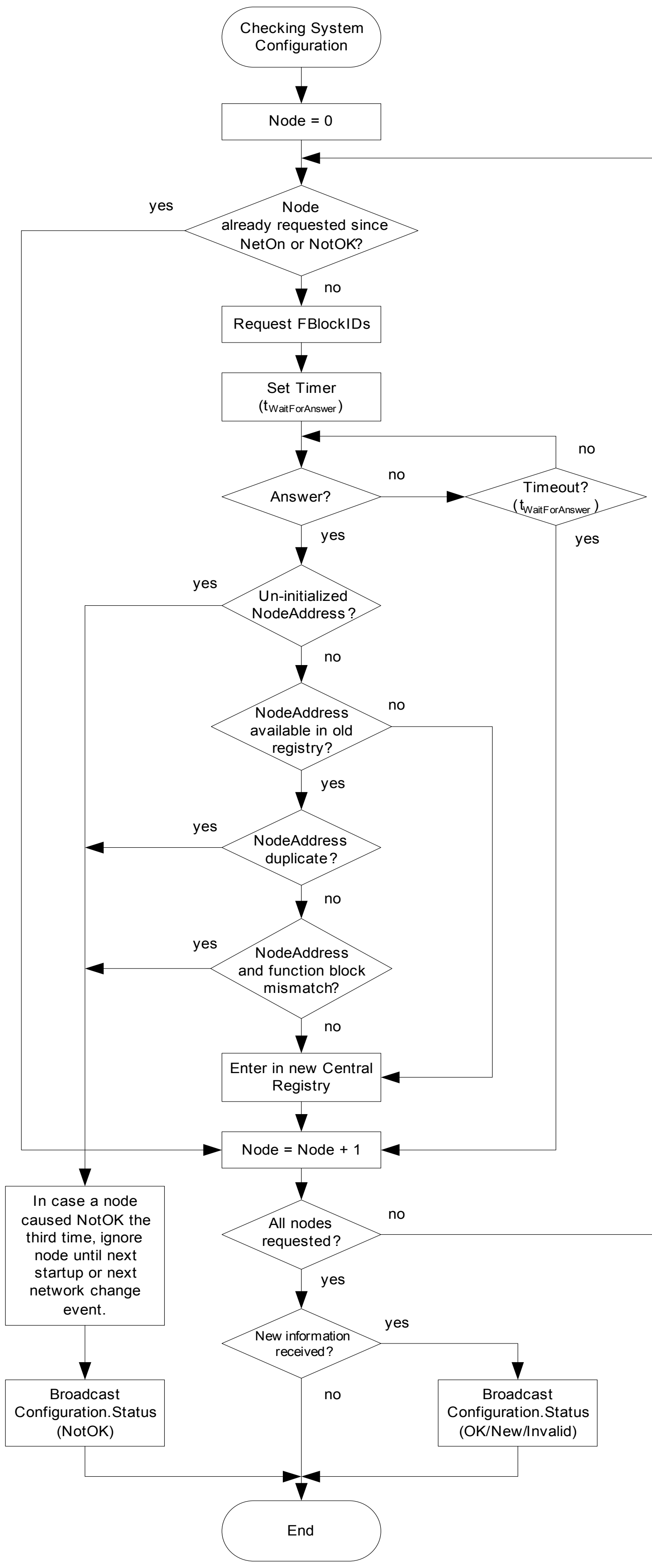

Checking System Configuration
Node = 0
Node already requested since NetOn or NotOK?
yes
no
Request FBlockIDs
Set Timer ($t_{WaitForAnswer}$)
Answer?
no
Timeout? ($t_{WaitForAnswer}$)
no
yes
yes
Un-initialized NodeAddress?
yes
no
NodeAddress available in old registry?
no
yes
NodeAddress duplicate?
yes
no
NodeAddress and function block mismatch?
yes
no
Enter in new Central Registry
Node = Node + 1
In case a node caused NotOK the third time, ignore node until next startup or next network change event.
All nodes requested?
no
yes
New information received?
yes
no
Broadcast Configuration.Status (NotOK)
Broadcast Configuration.Status (OK/New/Invalid)
End

Figure A-5-2: Flow in NetworkMaster during requesting system configuration

5.2 Network Slave Section

The flow in Figure A-5-3 shows how a Network Slave behaves during System Startup and when receiving Configuration.Status messages.

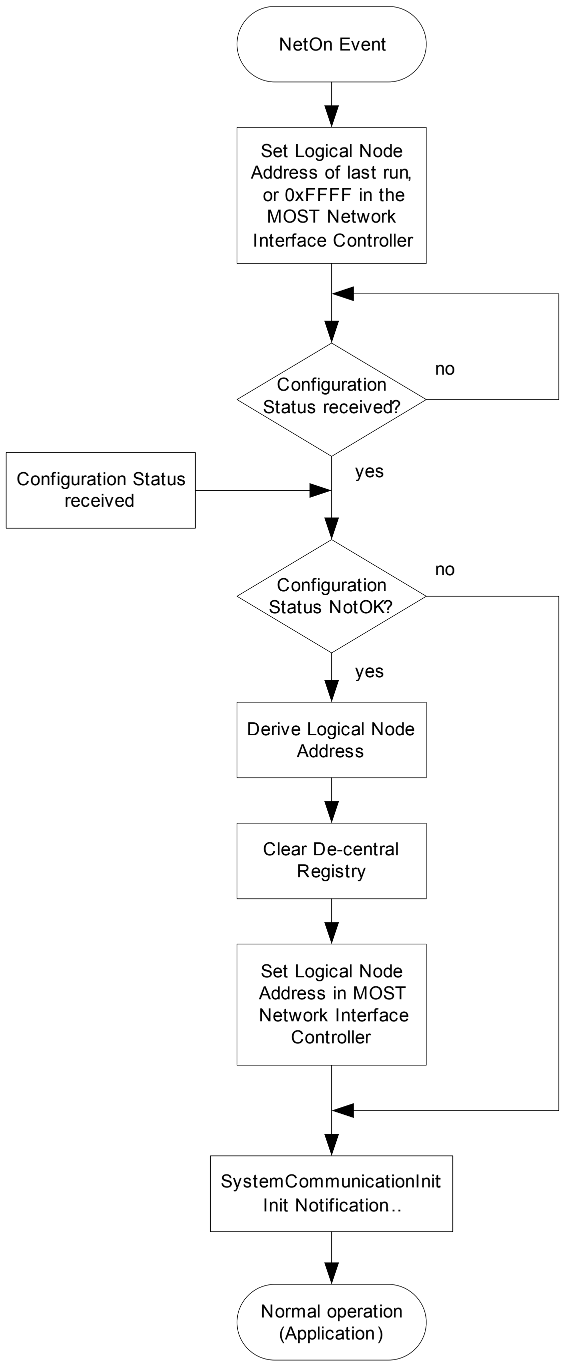

NetOn Event
Set Logical Node Address of last run, or 0xFFFF in the MOST Network Interface Controller
Configuration Status received?
no
yes
Configuration Status received
Configuration Status NotOK?
no
yes
Derive Logical Node Address
Clear De-central Registry
Set Logical Node Address in MOST Network Interface Controller
SystemCommunicationInit Init Notification..
Normal operation (Application)

Figure A-5-3: Flow of initialization on application level in a Network Slave

6 Appendix B: Synchronous Data Types

This appendix holds information of different synchronous data types.

Note: This information is only relevant until the "MOST Multimedia Streaming Specification" is released.

Data type 0x00 Audio:

Here, the additional parameters Resolution, AudioChannels, Delay and Handle/Channels are specified.

```
DataDescription = Resolution, AudioChannels, SrcDelay, Channels
```

Parameter **Resolution** (1 Byte) specifies the resolution of audio samples in Bytes. Parameter **AudioChannels** (1 Byte) specifies the number of audio channels, e.g., 1 for mono, 2 for stereo etc. Parameter **SrcDelay** (1 Byte) specifies the delay of the synchronous data with respect to the Timing Master. Each MOST Network Interface Controller keeps track of the mode delay (refer to section 3.1.5.3 on page 120). In the last parameter, the single **Channels** (1 Byte per channel) are listed. The first channel corresponds to the handle. If the source has not allocated channels at the moment, it returns 0xFF.

The MOST Network Interface Controller is able to receive many different audio formats and convert them to its raw data format, or to generate many different audio formats from the transported raw data. For audio transmissions, the following minimum appointments are valid:

- Audio-NF will be transported CD-DA compatible (Compact Disk Digital Audio)
- The sequence of channels is: Front left, front right, rear left, rear right. The most significant Byte is transmitted first.

Examples:

16 Bit Stereo:	Resolution = 0x02, Channels = 0x02, Sequence on the bus: MSB left, LSB left, MSB right, LSB right.
24 Bit Stereo:	Resolution = 0x03, Channels = 0x02, Sequence on the bus: MSB left, central Byte left, LSB left, MSB right, central Byte right, LSB right.

If the property SourceInfo is not implemented by the source, data type audio is assumed by default.

Data type 0x01 CD ROM:

This data type describes CD-ROM raw data before being processed by a CD-ROM decoder. This data might be of type audio, or CD-I, or Video-CD respectively.

```
DataDescription = Blockwidth, Channels
```

For data type CD-ROM, parameter “Blockwidth” is transmitted. It specifies the number of transmitted Bytes per MOST frame.

Examples:

Single Speed CD: Blockwidth = 0x04
Double Speed CD: Blockwidth = 0x08

Per Default, Blockwidth = 0x04 will be assumed.

7 Appendix C: List of Figures

8 Appendix D: List of Tables

INDEX

Notes:

Notes:

Notes: