MOST

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

Multimedia and Control
Networking Technology

MOST Specification
Rev 2.5
10/2006

Legal Notice

COPYRIGHT

LICENSE DISCLAIMER

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

CONTENT AND LIABILITY DISCLAIMER

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

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

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

FEEDBACK INFORMATION

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

TRADEMARKS

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

SUPPORT AND FURTHER INFORMATION

For more information on the MOST technology, please contact:

MOST Cooperation
Administration
D-76185 Karlsruhe
Germany

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

MOST®
COOPERATION

Contents

Document History

In general, “Section” refers to the current document, unless deletions are referred to. Sections that were deleted and not replaced by others are marked ~~strikethrough~~. If deleted sections were replaced by others, details about the substitution can be found in the change description.

Changes MOST Specification 2V5-00 (08/2006) to MOST Specification 2V5-00 (10/2006)

Change Ref.	Section	Changes
2V5_2_001	General	Fixed minor clerical errors.
2V5_2_002	1.3	Removed superfluous “Note 1” in MOST Document Structure diagram.
2V5_2_003	2.3.2.2	Corrected ranges in FBlockIDs table: Supplier Specific FBlockIDs range from 0xF0 to 0xFB (previously 0xF0 to 0xFE); 0xFD and 0xFE are in the reserved range.
2V5_2_004	3.8	Removed inequality $t_{WaitForAnswer} \le t_{DelayCfgRequest1}$. Max. Value for $t_{WaitForAnswer}$ increased from 500 to 700 ms.

Changes MOST Specification 2V4-00 to MOST Specification 2V5-00

Change Ref.	Section	Changes
2V5_001	General	Optical Interface changed to Physical Interface.
2V5_002	General	Removed old section 3.1.2 ‘Source Data Bypass’ MOST25 (up to MOST Specification Rev2.4) permitted that each MOST device could be switched from active (source data bypass disabled) to passive (source data bypass enabled) to reduce the source data delay in the network. Disabling the source data bypass in a MOST25 device adds a delay of 2 network frames to streaming source data. Therefore network delay compensation is useful for noise canceling, speech recognition, and multi-channel sound applications. The bypass to reduce the network delay, as well as the delay compensation is not necessary when using MOST50. The effective delay is negligible (approx. 300 ns per node). Starting with MOST Specification Rev2.5 all devices are assumed to be “active”. This section was removed because the Source Data Bypass was not used with MOST25 and it is no longer applicable for MOST50.
2V5_003	General	“light” changed to “modulated signal”
2V5_004	General	“FOT” changed to “Physical Interface”
2V5_005	General	Corrected grammar and spelling mistakes. Fixed cross-references for diagrams that were moved to other chapters (3-13, 3-14, 3-15). Unified timer names (e.g. occurrences of t_Diag_Slave were replaced by $t_{DiagSlave}$).
2V5_006	General	Unified spelling of MOST terms like NetworkMaster, ConnectionMaster, etc. Replaced “NetOn event” with “Init Ready event”; For MOST50, the “System Lock flag” was introduced.
2V5_007	General	Harmonized terminology: Source data, streaming data, packet data vs. synchronous/asynchronous data/area. Substituted “all-bypass” with “bypass”.
2V5_008	Glossary	Added Glossary chapter for introduction and definition of frequently used terms.
2V5_009	1.1	Revised introduction.
2V5_010	1.4	Rephrased remark about references to improve intelligibility. Added HTTP 1.1 RFC to referenced documents.
2V5_011	1.5	Added speed grade and physical interface differentiations.
2V5_012	2.1, 2.1.2, 2.1.3, 2.2.3	Improved wording, eliminated redundancies.
2V5_013	2.1.4	Removed "60 bytes". Refer to the Boundary Descriptor rather than description here. Added reference to bandwidth management section.
2V5_014	2.2.1	Added remark about support for different physical interfaces.

Change Ref.	Section	Changes
2V5_015	2.2.1.1	Substituted description of “Slave, Controller, HMI” with a more precise explanation.
2V5_016	2.2.5	Rephrased paragraph on notifications to promote clarity and consistent use of terminology.
2V5_017	2.2.6	Removed statement on use of functions that did not exist during development.
2V5_018	~~2.2.9~~	Deleted sections on delegation, heredity and device hierarchy.
2V5_019	2.3.1	Removed duplicate introductory part (FBlock grouping and access) that is already contained 2.2.1.2 and referenced from this section.
2V5_020	2.3.2.2	Changed reserved and proprietary FBlockID ranges, added table “Responsibilities for FBlockID and FktID ranges”. Added remark that Secondary Nodes apply to MOST25 only. Added note that Supplier specific FBlocks are not reported in NetBlock.FBlockIDs.Status.
2V5_021	2.3.2.4	Added the following: Supplier specific functions are not reported in <FBlockID>.FktIDs.Status. Notification.Set(SetAll) does not affect Supplier specific functions while on the other hand Notification.Set(ClearAll) clears Notification for Supplier specific functions. Added table “Responsibilities for FBlockID and FktID ranges”
2V5_022	2.3.2.3.6	Added description of the exceptional cases when wildcards can be used in replies. Limited “Secondary Node” error to MOST25.
2V5_023	2.3.2.5	Correction in OPType table (ErrorAck is not available for Properties and was removed).
2V5_024	2.3.2.5.1	Note added about segmented error messages. Added ErrorCode and ErrorInfo ranges for use by System Integrators and Suppliers. Added remark that “Method Aborted” must be sent, even if a method is not executed anymore. Moved example for sensor failure (Error Code 0x41) to section 2.3.12. Added remark that “Segmentation Error” might also occur in single telegram context. Added explanation of different ErrorCode and ErrorInfo positions for Error and ErrorAck. Differentiated between “Secondary Node” error for MOST25 and MOST50. Removed reference to “generally broken device” for the “Device Malfunction” error.
2V5_025	2.3.2.7.10	Added ranges for System Integrators and Suppliers in String identifier table. Added string type SHIFT_JIS (code 0x07). Reserved range therefore now starts with 0x08.
2V5_026	2.3.6.1	Substituted the abstract description of involved devices with an explanation of the setup for the example.
2V5_027	2.3.11	Entire section: Reordered OPType tables so that OPTypes appear in order of OPType ID. Removed common description (for methods and properties) from single parameter properties section (2.3.11.1) and inserted it here. Also added Trigger Method, Sequence Method, Sequence Property and Map to list of function classes.
2V5_028	2.3.11.1.1	Added footnote that parameters marked boldface are explained in detail.
2V5_029	2.3.11.1.2	Added new category “Data” to unit table. Added constants for Byte, kByte, and MByte. Renamed category “Volume” to “Miscelleaneous”. Added “%” (percent). Redefined category “Temperature” as “ Temperature and Pressure”. Added K, bar, and psi. Redefined category “Speed” as “Speed and Acceleration”. Added cm/s, °/s, and m/s^2.
2V5_030	2.3.11.1.5	Added description of DataType parameter.
2V5_031	2.3.11.1.7	Removed “Classified Stream” Parameter for Get OPType.
2V5_032	2.3.11.2.1, 2.3.11.2.2	Added function class Container to IntDesc Table. Removed reference to “internal OPTypes”.
2V5_033	2.3.11.2.3	Entire Function Class DynamicArray section exchanged.
2V5_034	2.3.11.2.4	Entire Function Class LongArray section exchanged.
2V5_035	2.3.11.2.5	Added new section: Function Class Map.
2V5_036	2.3.11.2.6	Corrected list of OPTypes in description of IntDescX.
2V5_037	2.3.11.3	Explained Sequence Method more precisely.
2V5_038	2.3.11.3.2	Added AbortAck OPType and SenderHandle parameter, which was missing for some OPTypes.

Change Ref.	Section	Changes
2V5_039	2.3.12	Described double registration more precisely: No second entry is generated in the Notification Matrix. Added description of behavior for invalid target addresses and effect in combination with groupcast addresses. Added list of reactions on system events. Added sensor failure example, previously under 2.3.2.5.1, and made it more generic (i.e., removed references to CAN).
2V5_040	3.1.1	Removed 2nd paragraph. Added bypass differentiation for oPhy and ePhy.
2V5_041	3.1.2	Removed 2nd paragraph.
2V5_042	3.1.3	Revised section.
2V5_043	3.1.3.1	Replaced section with modified MOST25 frame description and new MOST50 parts.
2V5_044	3.1.3.1.2	Added MOST50 Dynamic Boundary description, revised MOST25 description and added comparison table. Added note that streaming connections must be re-built after Boundary Descriptor change.
2V5_045	3.1.3.2	New section heading is “Control Data Transport”. Added minimum/maximum number of frames for MOST50.
2V5_046	3.1.3.3.1	Renamed section to “Distinction between Source Data and Control Data”.
2V5_047	3.1.3.3.4	Rephrased to match both MOST25 and MOST50.
2V5_048	3.1.3.3.5	Changed heading from “Synchronous Area” to “Streaming Data”. Added MOST50 description.
2V5_049	3.1.3.3.6	Changed heading from “Asynchronous (Packet Data) Area” to “Packet Data”. Added note on limited applicability for MOST50.
2V5_050	3.1.3.4.2	Restructured and separated MOST25 part from general description. Added paragraph on MOST50 frame rate and message rate.
2V5_051	3.1.4.1	Added address range 0x1000...0xFFFE “Reserved for future use” Changed range 0x440...0x4FF to “Reserved”. Removed last row in Address range table (“highest nibble reserved for future use”). Substituted “Unique node address” with “Logical node address” for consistency. Added distinction between dynamic and static logical node addresses.
2V5_052	3.1.4.3	New section ‘Automatic Allocation Mechanism’. Merged former sections 3.1.5.4 ‘Automatic Channel Allocation’ and 3.1.5.5 ‘Detection of Unused Channels’ into this section. Added MOST50 description, separated from MOST25 part.
2V5_053	3.2.1	Added description of Primary and Secondary Nodes concept. Removed names of application states; they are not relevant to the specification.
2V5_054	3.2.2.2	Revised the 2nd bullet. Included more general description of Init Ready event and System Lock flag.
2V5_055	3.2.2.2	Changed diagrams so that MOST25 and MOST50 are covered.
2V5_056	3.2.2.3 3.2.5.2	Moved and revised parts of sections ‘NetInterface Normal Operation’ and ‘Unlock’.
2V5_057	3.2.2.4	Added ring break diagnosis results table. Separated paragraph with applicability limited to MOST25. Improved diagrams and distinguished between MOST25 and MOST50. Removed remark on evaluation of signal during NetOn.NetInterfaceNormalOperation.
2V5_058	3.2.3	Merged former sections 3.10 and 3.3.3.2.6 ‘Secondary Nodes’ into this section. Tagged entire section as “MOST25 only” because MOST50 does not support Secondary Nodes.
2V5_059	3.2.4.1	Replaced AbilityToWake by PermissionToWake and CapabilityToWake.
2V5_060	3.2.5	Revised whole section ‘Error Management’.

Change Ref.	Section	Changes
2V5_061	3.2.5.5	Changed heading from ‘Low Voltage’ to ‘Supply Voltage’
2V5_062	3.2.5.6	Replaced “This recommendation is applicable…” with “This section applies…”
2V5_063	3.2.5.6.2	Mentioned $t_{WaitAfterOvertempShutDown}$ as relevant timer for restart attempts. Added note that the PowerMaster must support either re-start by its own decision or by user request.
2V5_064	3.3.1.1.1, 3.3.3.2.3, 3.3.3.3, 3.3.3.3.2	Removed parts that refer to a stored Central Registry.
2V5_065	3.3.1.2.3	Deleted parts of the section (device failure handling).
2V5_066	3.3.2	Added Shutdown.Start(Execute) transition from System State OK to NotOK. Added footnote stating that logical node addresses will not be recalculated for that transition.
2V5_067	3.3.3.3.2	Content of “Stable Network” section included here because it is was not relevant to 3.3.3.3.1, where a similar description is already referenced.
2V5_068	3.3.3.5.4	Clarification: New InstID is set by using the logical node address.
2V5_069	3.3.3.6.1	Added remark that Configuration.Status(NotOk) must be sent after ConnectionMaster becomes unregistered.
2V5_070	3.3.3.6.3	Added remark that Configuration.Status(New) with an empty list can only be sent after completion of the scan.
2V5_071	~~3.3.3.8~~	Removed “Verification Scan” and “Missing Devices” sections because the concept of a stored Central Registry was abandoned.
2V5_072	3.3.4.1.3	Changed description of conditions under which the Decentral Registry is cleared.
2V5_073	~~3.3.4.1.4~~	Removed “Persistence of the Decentral Registry” section.
2V5_074	3.3.4.3.3	Added recommendation not to include FBlocks with InstIDs derived from position in NetBlock.FlockIDs.Status or Central Registry.
2V5_075	3.3.4.3.7	Added note stating that the device also assumes System State NotOk when the NetInterface enters normal operation.
2V5_076	3.3.4.3.8	Removed Configuration.Status(Invalid, <empty>) from the list of examples.
2V5_077	3.3.4.3.15	Added description of reinitialization behavior regarding streaming/packet connections and notifications.
2V5_078	3.4.1	Emphasized that the example where address ranges are assigned according to device functionality relies on the static address range. Added remark that group address can be stored optionally. Improved the description of group address handling in case of device power loss.
2V5_079	3.4.3	Removed reference to “Register” to make the description more generic.
2V5_080	3.4.4	Removed section on High Level Retries. These are implemented according to the System Integrator’s own policy. As a result of the deletion, previous section 3.4.5 “Basics for Automatic Adding of Device Address” has become 3.4.4. Heading numbers of the following level 3 headings under 3.4 have decreased by one.
2V5_081	3.4.6.2	Added remark that telegram length has to be exact.
2V5_082	3.5.1	Revised section to a more general description regarding data and channels.
2V5_083	3.5.2.2.1, 3.5.2.2.3	Revised to better differentiate between MOST25 and MOST50.
2V5_084	3.5.2.2.4	New section “Order of Streaming Channel Lists” created.
2V5_085	3.5.2.3	Revised section.
2V5_086	3.5.2.2.1	Added footnote about SourceConnect. Added footnote that SourceActivity is optional. Revised section to a more general description.
2V5_087	3.6	More generalized description for MOST 25 priorities, previously under 3.6.1.1.

Change Ref.	Section	Changes
2V5_088	3.6.1	Removed previous section 3.6.1 on “Direct Access to the MOST Network Interface Controller”. Revised section (previously 3.6.2) to a more general description regarding different types of telegrams.
2V5_089	3.6.1.1	Added examples for different telegram types.
2V5_090	3.7	Former section 3.7 Managing Synchronous/Asynchronous Data removed. The content was already covered under the ConnectionMaster section. Previous section 3.8 “Connections” now resides under 3.7.
2V5_091	3.7.1	New section “Bandwidth Management”, containing description of Boundary Descriptor adjustment.
2V5_092	3.7.2.2	Specified that a source shall perform 20 retries before Allocate is considered failed by the ConnectionMaster.
2V5_093	3.7.2.4.2	Slightly restructured and rephrased.
2V5_094	3.7.2.1	Revised.
2V5_095	3.8	Updated description of timer t_{Bypass}. Changed description of $t_{SlaveShutdown}$ to improve intelligibility. Revised description of $t_{DelayCfgRequest1}$ and $t_{SlaveShutdown}$. Added timer $t_{WaitAfterOvertempShutDown}$, $t_{WaitForNextSegment}$ and $t_{MsgResponse}$.. Renamed t_{Diag_Light} to t_{Diag_Signal}. Added $t_{WaitForProperty}$ note on overwriting default maximum value by FBlocks.
2V5_096	4	Replaced entire chapter with content from WG Phys. Layer.
2V5_097	5.1.1	Previous section 5.1.1 “System Startup when a Central Registry is Available” removed because the concept of a stored Central Registry has been abandoned. Section 5.1.1 now contains “Flow of System Initialization Process by the NetworkMaster”. Replaced “old registry” with “existing registry” in Figure A-5-2.
2V5_098	Appendix B	Removed previous Appendix B “Synchronous Data Types”. The contained information is now part of the “MOST Multimedia Streaming Specification”. Examples for typical data rates for MOST25 and MOST50 are now contained in Appendix B.
2V5_099	Document History	Added remark to emphasize that section references are based on the current document. Moved document history for versions prior to 2.5 to the end of the document.

Glossary

Term	Definition
Boundary Descriptor	The Boundary Descriptor determines the amount of bandwidth available between Streaming Data and Packet Data.
Central Registry	Contains a lookup-table for cross-referencing logical and functional addresses. Implemented in the NetworkMaster.
ConnectionMaster	Streaming connections are managed by the ConnectionMaster.
Connection Label	In MOST50 Connection Labels are used to identify streaming connections.
Control Data	Data packets containing control information.
Decentral Registry	Contains a lookup table for cross -referencing logical and functional addresses. Implemented in each node.
Device	A MOST device is a physical unit, which can be connected to a MOST network via a MOST Network Interface Controller.
DeviceID	The DeviceID stands for a physical device or a group of devices in the network. The DeviceID (RxTxAdr) can represent a node position address (RxTxPos), a logical address (RxTxLog) or a group address.
FBlock	A function block. Function blocks group functions that are particular to a specific application, for example radio or telephone.
FBlock Shadow	The FBlock Shadow is an implementation concept where an FBlock is controlled through a proxy for that particular FBlock.
Frame	Data transfer is organized in frames. Frames consist of administrative data and payload.
Function	A function is a part of a function block through which it communicates with the external world.
Init Ready event	When the system is in a stable lock state (i.e., operational), the Init Ready event is fired.
Method	Function that can be started and which leads to a result after a certain period of time.
NetInterface	The expression NetInterface stands for the entire communication section of a node: the physical interface, the MOST Network Interface Controller, and the Network Service.
NetOn State	The NetOn state corresponds to the “NetInterface normal operation” state.
NetworkMaster	The NetworkMaster controls the System State and administrates the Central Registry.
Node	A node is characterized by the existence of one NetInterface.
Packet Data	Asynchronous data packets, such as IP packets.
PowerMaster	The PowerMaster is responsible for power management throughout the MOST system.
Property	Function for determining or changing the status of a device.
RxTxAdr	Either a logical node address (RxTxLog), a node position address (RxTxPos), or a group address. For receiving nodes, it is called RxAdr, for sending nodes TxAdr.
RxTxLog	A logical node address. It must be unique in the system and is called RxLog for receiving nodes and TxLog for transmitting nodes.
RxTxPos	A node position address. The node position address is called RxPos for a receiving node and TxPos for a transmitting node. The node position address depends on the physical position of the MOST Network Interface Controller.
Source Data	The term Source Data combines Streaming Data and Packet Data.
Streaming Data	Content, such as audio or video data, which has to be transmitted synchronously or isochronously, is referred to as Streaming Data.
System Lock flag	This flag is only available for MOST50 and indicates that the TimingMaster has detected a stable lock of the system.
System Scan	The process of collecting information from the Network Slaves, performed by the NetworkMaster.
System State	There are two System States: OK and NotOK. In OK, the system is in normal operation mode, in NotOK it is being initialized or updated.
TimingMaster	The TimingMaster provides generation and transport of the system clock, frames, and blocks.

1 Introduction

1.1 Purpose

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

1.2 Scope

This document contains the specification of the 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.

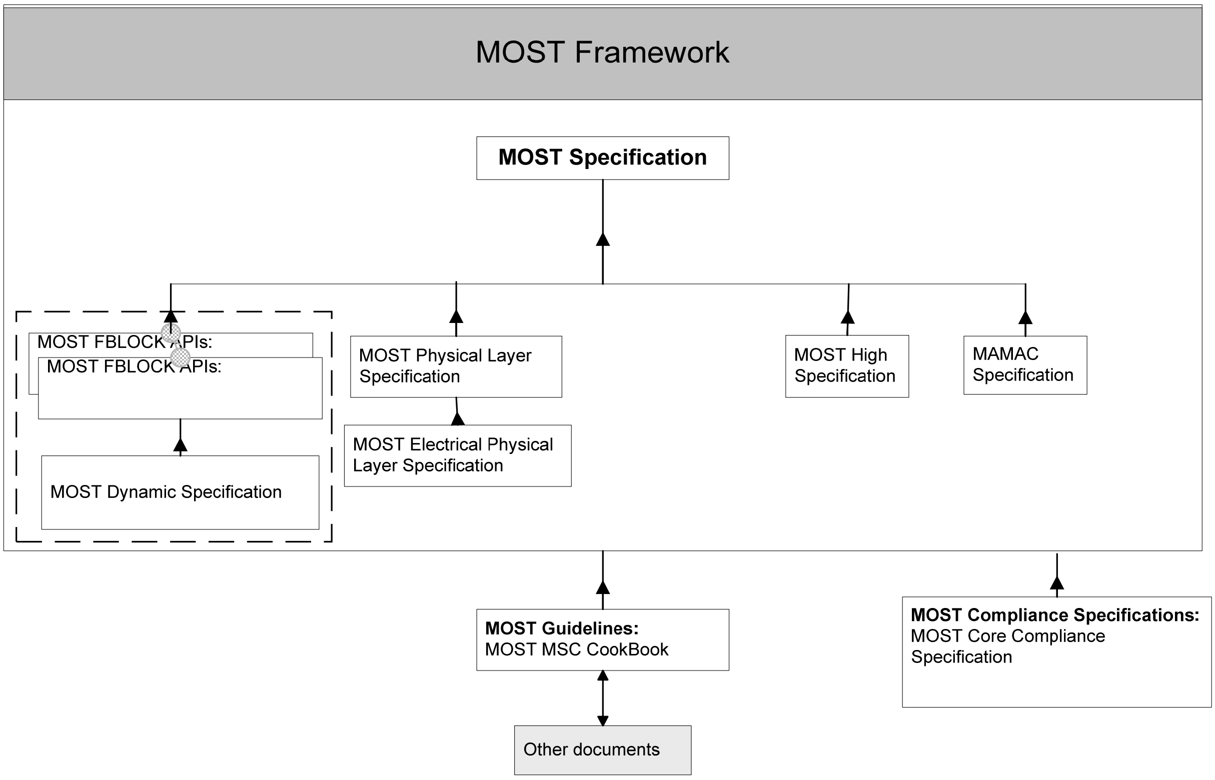

MOST Framework
MOST Specification
MOST FBLOCK APIs:
MOST FBLOCK APIs:
MOST Dynamic Specification
MOST Physical Layer Specification
MOST Electrical Physical Layer Specification
MOST High Specification
MAMAC Specification
MOST Guidelines:
MOST MSC CookBook
MOST Compliance Specifications:
MOST Core Compliance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

Figure 1-1: MOST Document Structure

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

1.4 References

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

Document	Revision
RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1	June 1999
-	-

Comment: The MOST Specification does not depend on other documents. All other documents that are part of the MOST Specification Framework can be accessed through the MOST Cooperation website.

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.

In some cases a differentiation between specific implementations (speed grade, physical interface) is necessary. The following terms are used to mark corresponding passages.

Term	Description
MOST25	25 MBit/s speed grade
MOST50	50 MBit/s speed grade
ePhy	Electrical Physical Layer
oPhy	Optical Physical Layer

2 Application Section

2.1 Overview of Data Channels

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

2.1.1 Control Channel

On the Control Channel, data packets are transported to specific addresses, as they are on the Packet Data 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 packet data area is specified for transmissions requiring high bandwidth in a burst-like manner.

2.1.2 Streaming Data Channel

Continuous data streams that demand high bandwidth (typically multimedia data, such as audio or video) are transported over the Streaming Data Channels. The connections are administered dynamically via the Control Channel.

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

2.1.3 Packet Data Channel

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

2.1.4 Managing Streaming/Packet Bandwidth

In a MOST Network the overall available bandwidth can be divided between the Streaming Channel and the Packet Data Channel in a flexible way by setting the Boundary Descriptor. The Boundary Descriptor can be modified by using the mechanism provided by the MOST Network Service.

The value of the Boundary Descriptor depends on the requirements of the system and can be changed dynamically. Supervision and changing of the bandwidth or position of the Boundary is done in the TimingMaster. The TimingMaster is responsible for forwarding the information about the Boundary's position to all nodes in the network. This task is handled automatically on the MOST Network Interface Controller level.
Please refer to 3.7.1 Bandwidth Management for a detailed description.

2.2 Logical Device Model

The following sections describe different kinds of devices. A MOST device is a physical unit that 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 that are called function blocks (FBlocks), for example, tuner, amplifier, or CD player. It is possible that there are multiple FBlocks in a single MOST device, such as a tuner and an amplifier combined in one case and connected to the MOST network via a common MOST Network Interface Controller. In addition to the FBlocks, which represent applications, each MOST device has a special FBlock called the NetBlock. The NetBlock provides functions related to the entire device. Between the FBlocks and the MOST Network Interface Controller, the Network Service forms an intermediate layer providing routines to simplify the handling of the MOST Network Interface Controller.

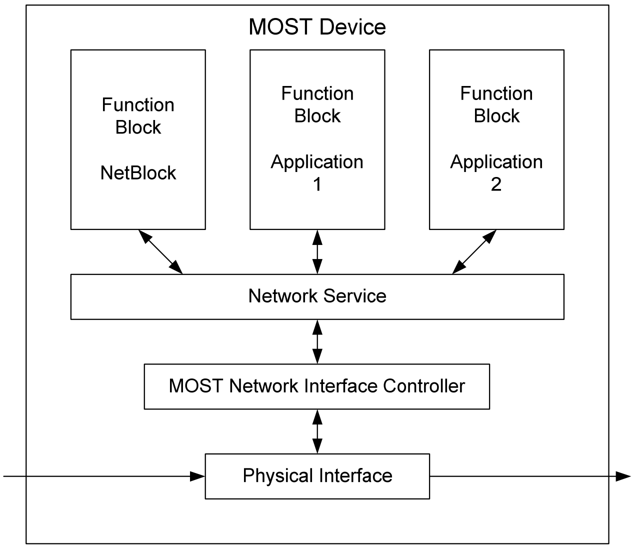

MOST Device
Function Block NetBlock
Function Block Application 1
Function Block Application 2
Network Service
MOST Network Interface Controller
Physical Interface

Figure 2-1: Model of a MOST device

The MOST standard supports different physical interfaces. These are described in the respective physical layer specifications.

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

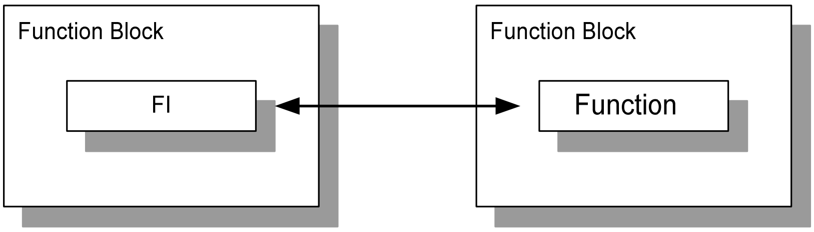


Figure 2-2: Communication with a function via its function interface (FI)

2.2.1.1 Slave, Controller, HMI

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

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

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

Note:
On application level, the Controller may use FBlock Shadows to control FBlocks.

2.2.1.2 First Introduction to MOST Functions

This section gives a brief introduction to the structure of MOST functions. This knowledge is necessary to understand the following examples. Chapter 2.3 on page 28 explains the structure of MOST functions in more detail.

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

FBlockID.FktID

When accessing functions, certain operations are applied to the respective function which is either a property of the FBlock or a method it provides. The type of operation is specified by the OPType. The parameters of the operation follow the OPType, resulting in the following structure:

FBlockID.FktID.OPType(Data)

2.2.2 Functions

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

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

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

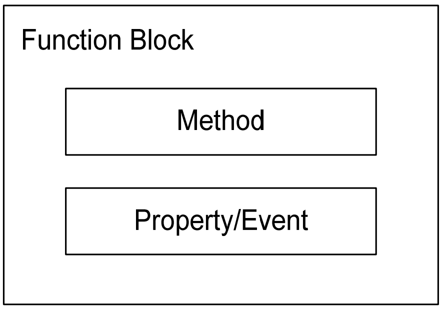


Figure 2-3: Structure of an FBlock consisting of functions classifiable as methods, properties and events

2.2.3 Methods

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

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

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

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

Controller	Slave
Start of a method with parameters (*Start/StartResult*)	Error with cause for error (*Error*) Execution report with results (Result) Intermediate result (Processing)

The respective MOST mechanisms needed for this messaging are described in depth in chapter 2.3 on page 28.

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 an FBlock, 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.

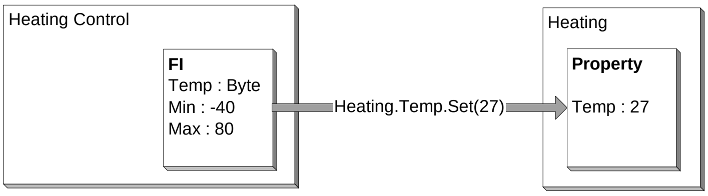


Figure 2-4: Setting a property (temperature setting of a heating)

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

2.2.4.2 Reading a Property

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

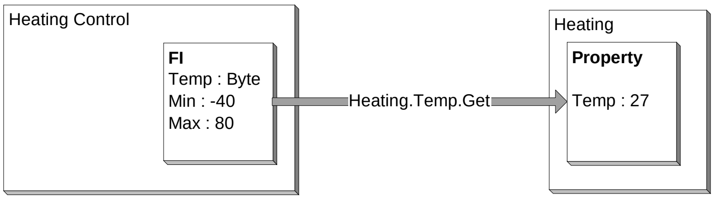


Figure 2-5: Reading a property (temperature setting of a heating)

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

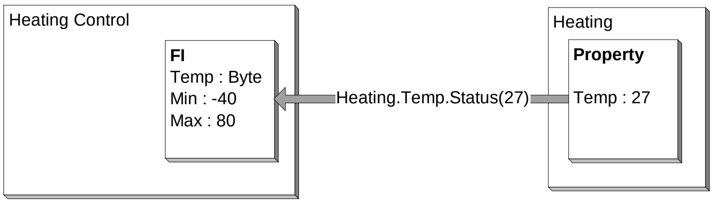


Figure 2-6: Status report of property temperature setting

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

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

The MOST functions needed for this messaging are described in depth in chapter 2.3 on page 28.

2.2.5 Events

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

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

Notifications can be used to indicate reaching of limits, the change of measured values in FBlocks (e.g., the play time of a CD player has changed), or in the HMI (e.g., reception of a new value of mileage received via a CAN gateway). Notifications are sent only to those FBlocks that have previously applied for the notification by sending an appropriate request to the Slave (and are therefore included in the Slave's Notification Matrix).
(Refer to chapter 2.3.12 on page 102).

2.2.6 Function Interfaces

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

To communicate with a function, a Controller or an HMI needs information about the available parameters, their limits, and the allowed operations (=FI).

In general, this information is available in the control device, and is encoded in the control program. The FI was passed on, for example, like a device specification. To simplify the exchange of FIs, especially between different manufacturers, a formal description may be used that can be exchanged between the developers of Slaves and Controllers, like the well-known header files in the C programming language.

The contents of the FIs are usually known during implementation of a device (well known functions). It is also possible that FIs are transported on the bus during runtime, making it possible to dynamically reconfigure a HMI.

Example:

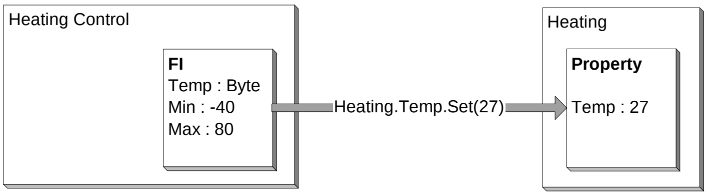


Figure 2-7: Example for a function interface (FI)

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 more information.

During operation it is possible that a FI changes dynamically. In that case, all the FBlocks that have subscribed to notification will get the new interface description through the notification mechanism. For more information about notification, please refer to section 2.3.12 on page 102.

2.2.7 Definition Example

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

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

(Pascal Syntax)

```
Device
  {
     TTuner Tuner;
     TNetBlock NetBlock;
  } MyTuner
```

(C Syntax)

The definition specifies that MyTuner contains an FBlock Tuner of type TTuner and an FBlock NetBlock of type TNetBlock.

TTuner is an FBlock and can be defined in the following way:

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

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

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

A special character at the beginning of a name can indicate the type of function (p = property, m = method, e = event).

Now property pStation will be defined as follows:

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

```
Property
  {
    long Frequency;
    Bool TP;
    Byte Quality;
  } TStation;
```

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

Now method mSearch will be defined:

```
TSearch = Method
            Up : Bool;
            Start : Long;
          end;
```

```
Method
  {
    Bool Up;
    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 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.

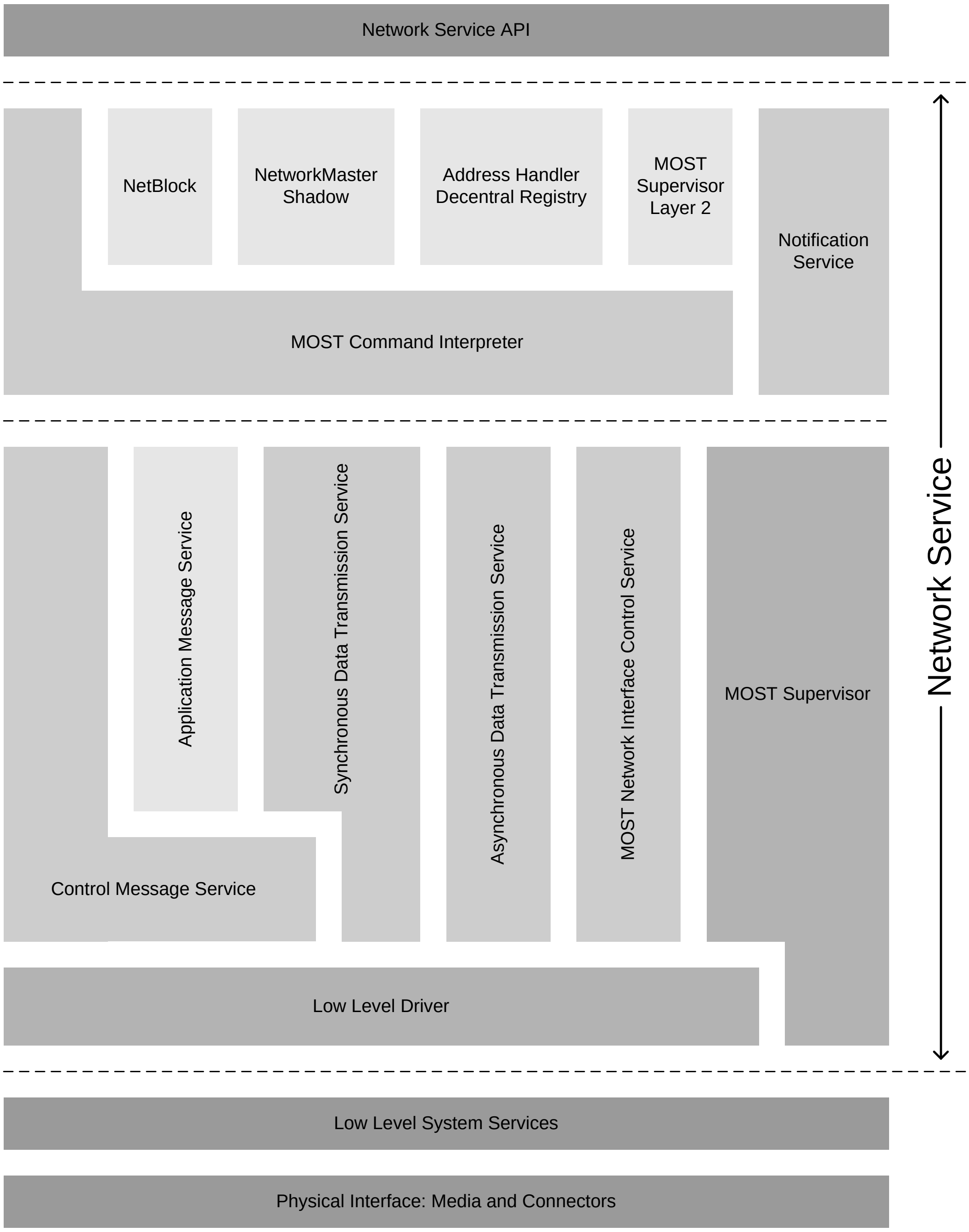

Network Service API
NetBlock
NetworkMaster Shadow
Address Handler Decentral Registry
MOST Supervisor Layer 2
Notification Service
MOST Command Interpreter
Application Message Service
Synchronous Data Transmission Service
Asynchronous Data Transmission Service
MOST Network Interface Control Service
MOST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

Figure 2-8: MOST Network Service

2.3 Protocols

2.3.1 Protocol Basics

As already described in section 2.2.1.2 on page 20, functions are addressed without considering the devices they belong to (on the application level). The resulting structure is this:

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 20, three components were added: DeviceID, InstID and Length. 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 of the sender.

2.3.2.2 FBlockID

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

"System Specific" proprietary FBlockIDs can be used by a System Integrator (e.g., a car maker). They are specific for a system and are coordinated by the System Integrator between the suppliers developing devices for this system. A second kind of proprietary FBlockIDs is called "Supplier Specific". Those FBlockIDs can be used by suppliers for any proprietary purpose. The special FBlockID 0xFF addresses all FBlocks within a MOST device, except the NetBlock. Since this can be regarded as a broadcast function, no error status messages should be returned.

The table below shows a collection (incomplete) 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 2-1: FBlockIDs (part 1)

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

Kind	FBlockID 8 Bit	Name	Explanation
Receiver	**0x4x**		
	0x40	AM/FM Tuner	
	0x41	TMCTuner	
	0x42	TV Tuner	
	0x43	DAB Tuner	
	0x44	Satellite Radio	
Communication	**0x5x**		
	0x50	Telephone	
	0x51	Phonebook	
	0x52	Navigation System	
	0x53	TMC Decoder	
	0x54	Bluetooth	
Video	**0x6x**		
	0x60	Display	
	0x61	Camera	
	0x62	Video Tape Recorder	
Reserved	**0x70 …0x9F**	**Reserved for future use**	
Proprietary			
	0xA0…0xC7	**System Specific**	
	0xC8	**Reserved**	
	0xC9…0xEF	**System Specific**	
	0xF0 …0xFB	**Supplier Specific**	
	0xFC	**Secondary Node**	**MOST25 only!**
	0xFD…0xFE	**Reserved**	
	0xFF	**All**	

Table 2-2: FBlockIDs (part 2)

The range between 0x00 and 0x9F can only be assigned by the MOST Cooperation.
Supplier specific FBlocks are not reported in NetBlock.FBlockIDs.Status.
Supplier specific FBlocks have to ignore all commands which contain FBlockID 0xFF.

2.3.2.3 InstID

There may be several equal[1] FBlocks (Instances) with the same FBlockID in the system (two CD changers, four active speakers, several diagnosis blocks, etc.). In order to address these FBlocks unambiguously, the FBlockID is complemented by an eight-bit instance identification number (InstID). The combination of 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 NetworkMaster 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 FBlock has InstID 0x01. In case there are several FBlocks of the same kind within one MOST device, the default numbering within the device starts at 0x01 and is then incremented. In principle, as long as the InstID provides the possibility to differentiate between equal FBlocks, the InstID can be chosen freely. For example, in static systems the System Integrator may choose to use hard coded InstIDs or set the InstIDs depending on certain ranges with respect to the supported functions of the FBlock.

Note: Wildcards must not be used for InstID assignment.

2.3.2.3.3 InstID of NetBlock

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

2.3.2.3.4 InstID of NetworkMaster

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

2.3.2.3.5 InstID of Function Block EnhancedTestability

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

---

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

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

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

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

b) The error message “Segmentation Error” (error code 0x0C) shall have the same InstID as the original message. This error is handled on transport layer but not on application layer, therefore the InstID is not evaluated.

MOST25
c) The error message “Secondary Node” (error code 0x0A) shall have the same InstID as the original message, as a Secondary Node never has an FBlock implemented.
This exception is an extension of use case “a”. If the request is aimed at the NetBlock, the error message “Secondary Node” shall contain the actual node position as InstID.

2.3.2.4 FktID

The FktID represents a function. This means a function unit (Object) within a device that provides operations that can be called via the network. Examples for functions are: play 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 FBlock. On the application level, the FktID is extended to 2 bytes. Exceptions to this rule will be explicitly marked.

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

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

2. **Mandatory (0x200...0x3FF)**
   Functions that are mandatory for the application of the FBlock, like the basic drive in all FBlocks 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 throughout the entire system!

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

6. **Proprietary/ Supplier Specific (0xF00...0xFFE)**
   Functions that can be used by suppliers for any proprietary purpose.
   - Supplier specific functions are not reported in <FBlockID>.FktIDs.Status.
   - If a supplier specific property supports notification, notification of this property is not affected by the <FBlockID>.Notification.Set(SetAll) command.
   - If a supplier specific property supports notification, notification of this property is cleared by the <FBlockID>.Notification.Set(ClearAll) command.

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

0x000	**FktIDs**	Reports the FktIDs of all functions contained in the FBlock (refer to section 2.3.9 on page 64).
0x001	**Notification**	Distribution list for events (refer to section 2.3.12 on page 102).
0x002	**NotificationCheck**	Check whether the distribution list for events is still as it should be.

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

- **Unique**

- **Coordination**

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

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

FktID / FBlockID	Coordination	Mandatory/ Extension	Unique	Proprietary/ System Specific	Proprietary/ Supplier Specific
MOST Co.	MOST Co.	MOST Co.	MOST Co.	System Integrator	Supplier
System Specific	MOST Co.	System Integrator	MOST Co.	System Integrator	Supplier
Supplier Specific	MOST Co.	Supplier	MOST Co.	Supplier	Supplier

Table 2-3: Responsibilities for FBlockID and FktID ranges

Please note:
Before using any proprietary function or proprietary FBlock, a Controller must verify the identity of the device. This can be done, for example, by reading the DeviceInfo property.

2.3.2.5 OPType

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

OPType	For Properties	For Methods
Commands:		
0	Set	Start
1	Get	Abort
2	SetGet	StartResult
3	Increment	Reserved
4	Decrement	Reserved
5	GetInterface	GetInterface
6	Not allowed	StartResultAck
7	Not allowed	AbortAck
8	Not allowed	StartAck
Reports:		
9	Reserved	ErrorAck
A	Not allowed	ProcessingAck
B	Reserved	Processing
C	Status	Result
D	Not allowed	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][1]	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	**MOST25 Secondary Node**	Return Address of Primary	Address of that node which is responsible for the Secondary Node sending the error.
	MOST50 Reserved	--	
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)

---

[1] ErrorAck requires a SenderHandle in Data[0] and Data[1]. This results in different ErrorCode and ErrorInfo positions for Error and ErrorAck.

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
		0xC0..0xEF	System Integrator specific
		0xF0..0xFE	Supplier specific
0x40	**Busy** Function is available, but is busy	--	No Info
0x41	**Not available** Function is implemented in principle, but is not available at the moment	--	No Info
0x42	**Processing Error**	--	No Info
0x43	**Method Aborted** This error code can be used to indicate, that a method has been aborted by the Abort / AbortAck OPTypes	--	No Info
0xC0..0xEF	**System Integrator (e.g., car maker) specific.**	Optional	
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 telegrams 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 telegrams. Individual error telegrams using error code 0x20 should be avoided if possible.

For avoiding infinite loops with respect to reporting errors, errors are reported only from Slave to Controller. In addition, no reply of error telegrams is allowed on reception of broadcast messages.

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

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

2. Application Error – Parameter Error:
The specified length does not match the actual length of the data field. There have been too few parameters, too many parameters, or one parameter is out of range. Parameter errors are reported by the ErrorCodes 0x05 and 0x06. Messages are only accepted when being completely correct. In particular, this means that the length of the parameter area must be correct. The only exception is the handling of arrays that are too short (refer to section 2.3.11.2 on page 77).

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.

MOST25

6. Application Error – Error Secondary Node:
Detailed information about Secondary Nodes is to be found in section 3.2.3 on page 136. 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 that the requested function is temporarily not available due to a device malfunction.

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, for example, by missing segments, wrong order of arrival or exceeding the timeout between two segments. In case of such an error, the parts of the message that have already been received are discarded. In addition, the application within the receiver is notified of the error by the Network Service.

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

The segmentation error notifies the sender about the failure of the transfer. Therefore, the sender’s application may react in an appropriate way, for example, by 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 telegrams can only be directed from the Slave to the Controller, to avoid infinite loops of error telegrams. 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 Network Service.

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.
The error report “Method Aborted” must always be sent back as a result to a successful abort request, even if the method is not executed anymore, for example, because it has finished successfully already.

The examination and processing of errors is done in the logical and temporary sequence as described above and in Figure 2-9 on page 40.

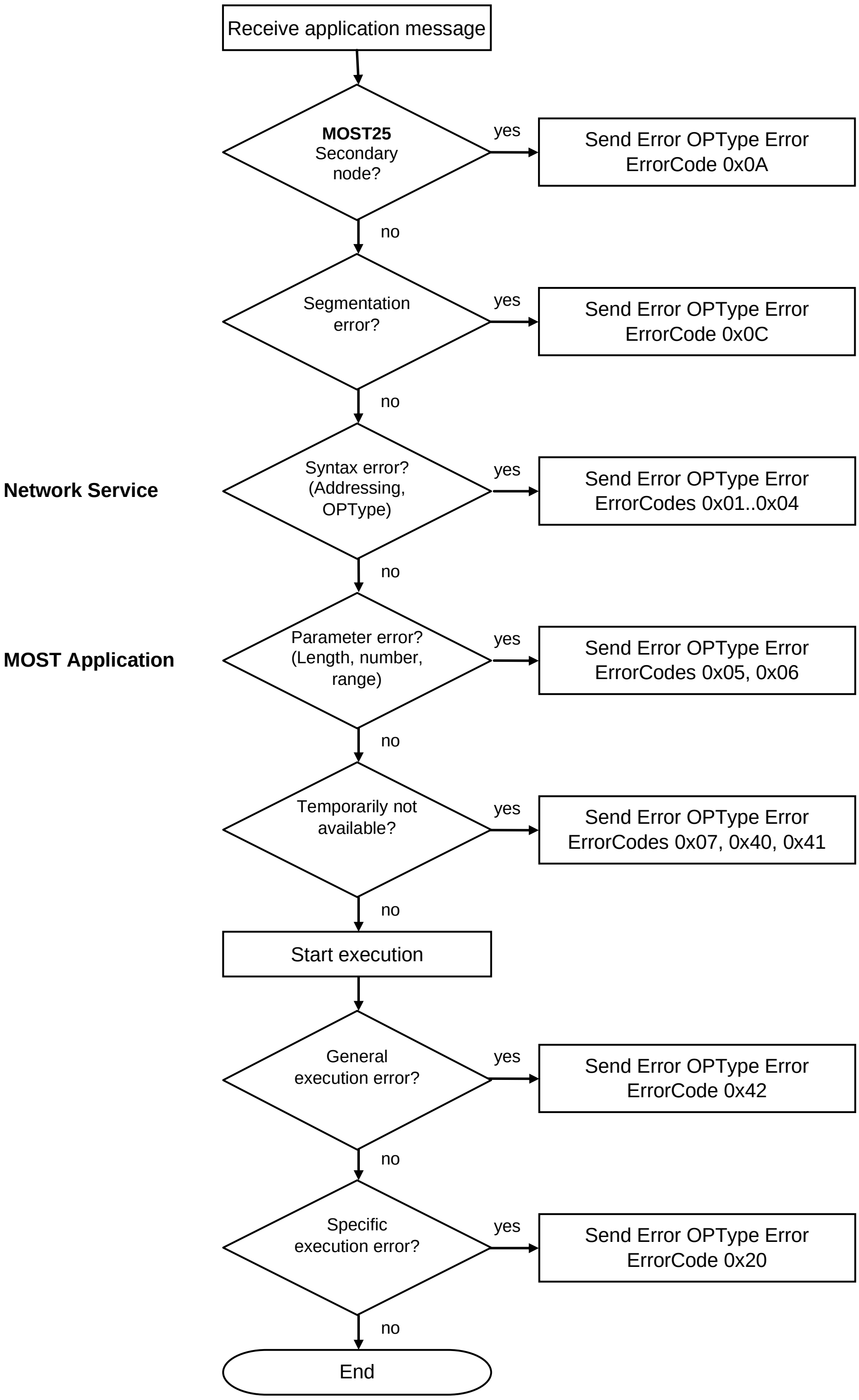

Receive application message
MOST25
Secondary node?
yes
Send Error OPType Error ErrorCode 0x0A
no
Segmentation error?
yes
Send Error OPType Error ErrorCode 0x0C
no
Network Service
Syntax error? (Addressing, OPType)
yes
Send Error OPType Error ErrorCodes 0x01..0x04
no
MOST Application
Parameter error? (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 2-9: 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.

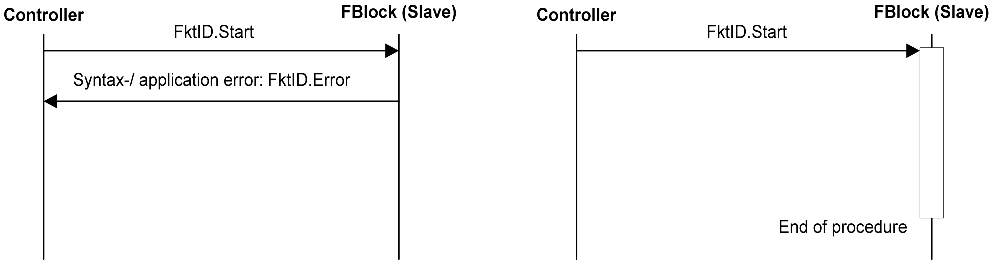


Figure 2-10: Sequences when using Start with and without error

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 with an error "Busy". The already running method is not affected by this new incoming request. For running several instances of the same method, StartAck 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.

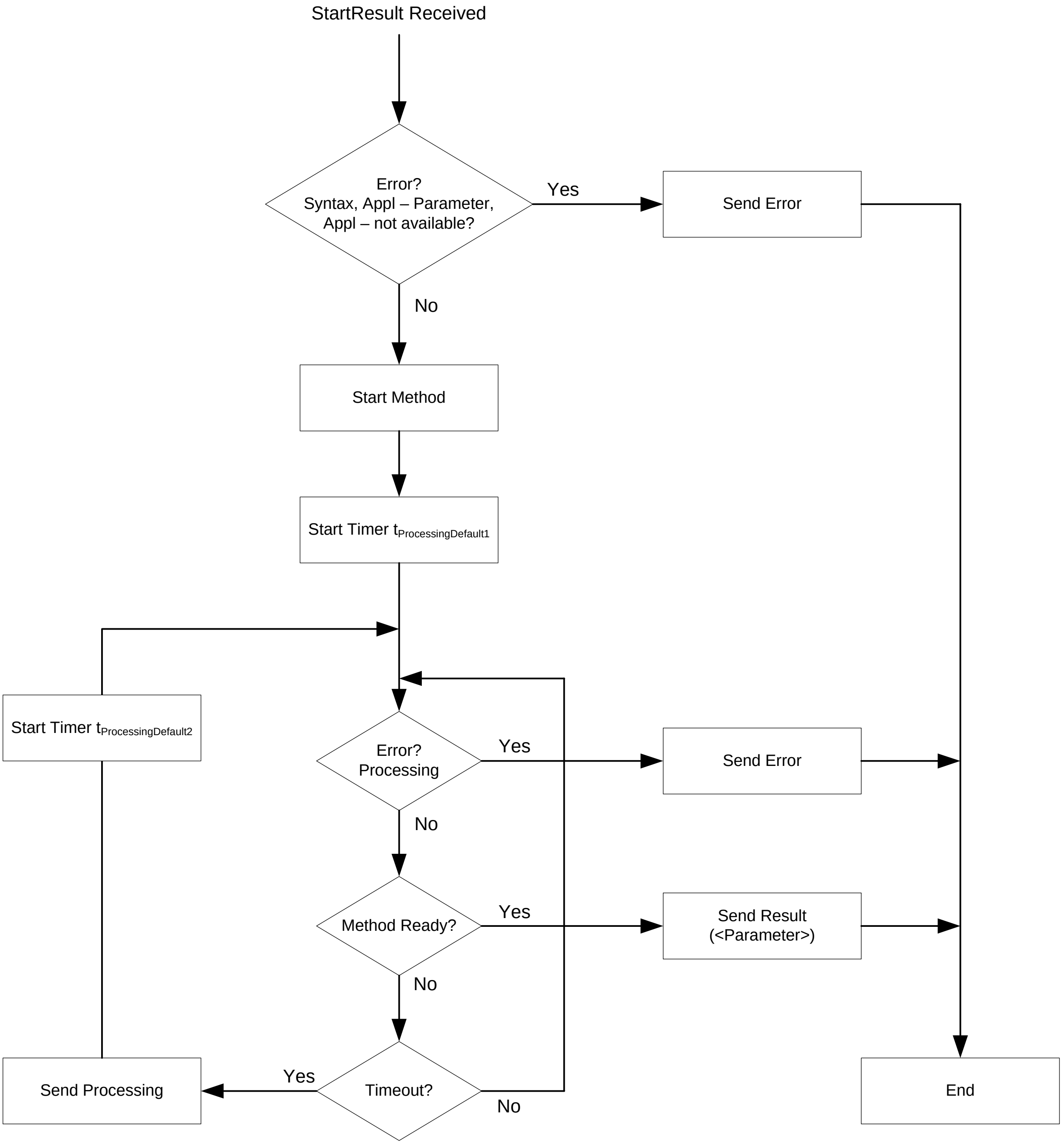


Figure 2-11: Flow for handling communication of methods (Slave's side)

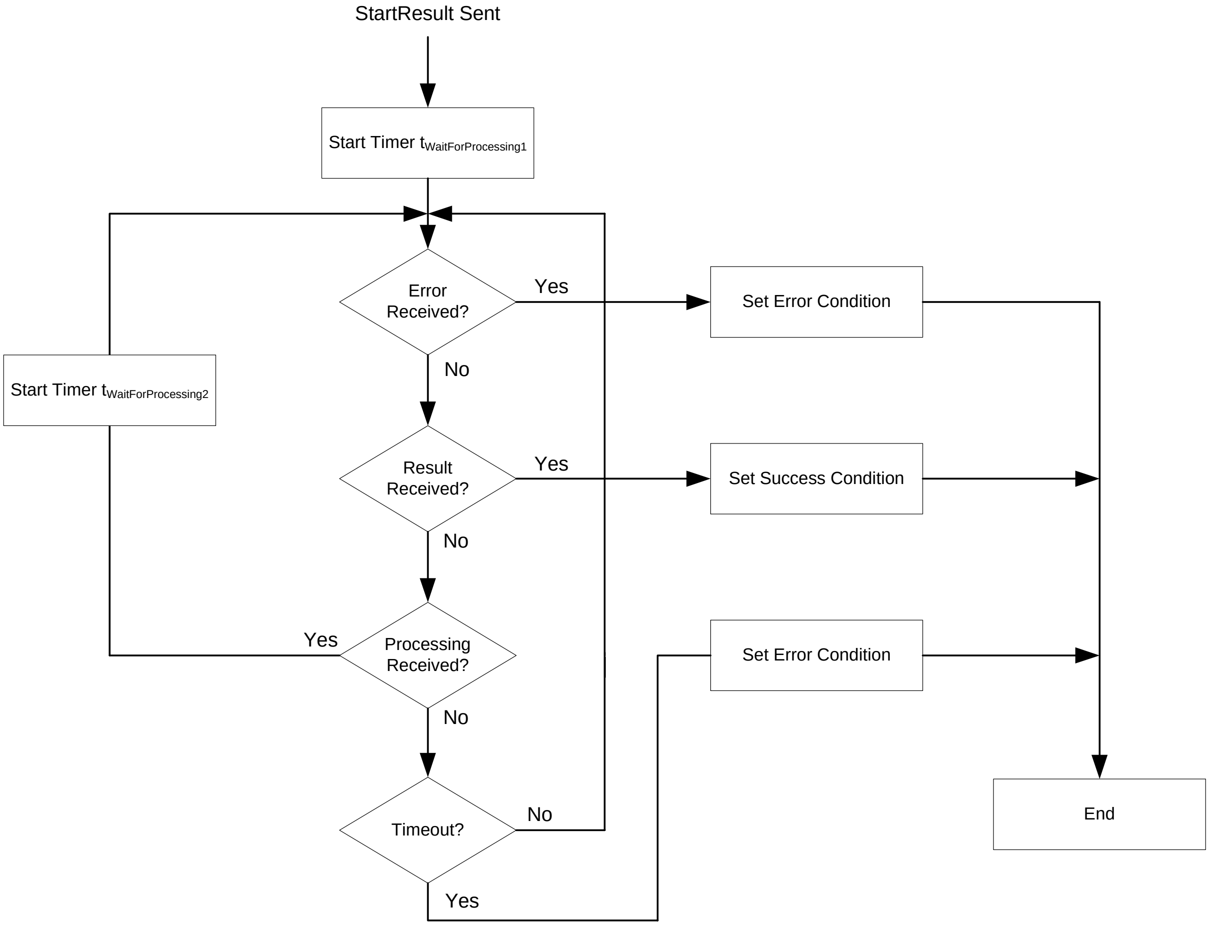


Figure 2-12: Flow for handling communication of methods (Controller's side)

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 41). The only difference is, that the first parameter transports the SenderHandle (refer to section 2.3.6.2 on page 62).

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 Get request, a reply using Status will be generated, if the syntax check has shown no errors. Otherwise, Error will be returned. A property shall reply 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, for example, 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 FBlock.

2.3.2.5.7 SetGet, Status, Error

SetGet is the preferred way of controlling FBlocks for Controllers:

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

SetGet is a combination of Set and Get, which means that the Controller (in case of a correct syntax) automatically gets the changed status in return. 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

These OPTypes are available for methods only. When used, Abort terminates the execution of a method. The message abortion is confirmed through an Error(Aborted) message. Abort must not have any parameters. Please note that methods in general should be aborted only by that application which has started the method. After the method has been aborted, information about this is sent out. Please see 9 Application Error – Method Aborted: on page 39 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 62). 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 39 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 bytes.

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. Longer protocols must be segmented, that is, 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
BitField
Unsigned Long
Signed Long
Short Stream
Unsigned Byte
Signed Byte
Enum
String
Unsigned Word
Signed Word
Stream
Classified 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 * 10Exponent

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 66). It is not transported together with the parameter.

2.3.2.7.1 Boolean

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

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 bytes	-

2.3.2.7.7 Signed Word

Definition of Type	Comments
2 bytes	-

2.3.2.7.8 Unsigned Long

Definition of Type	Comments
4 bytes	-

2.3.2.7.9 Signed Long

Definition of Type	Comments
4 bytes	-

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
0x07	SHIFT_JIS	No
0x08 - 0xBF	reserved	
0xC0..0xEF	System Integrator (e.g. Car Maker)	
0xF0..0xFF	Supplier	

Content: Characters

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

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

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

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

2.3.2.7.11 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 53). In most cases, only one instance of the FBlock 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.

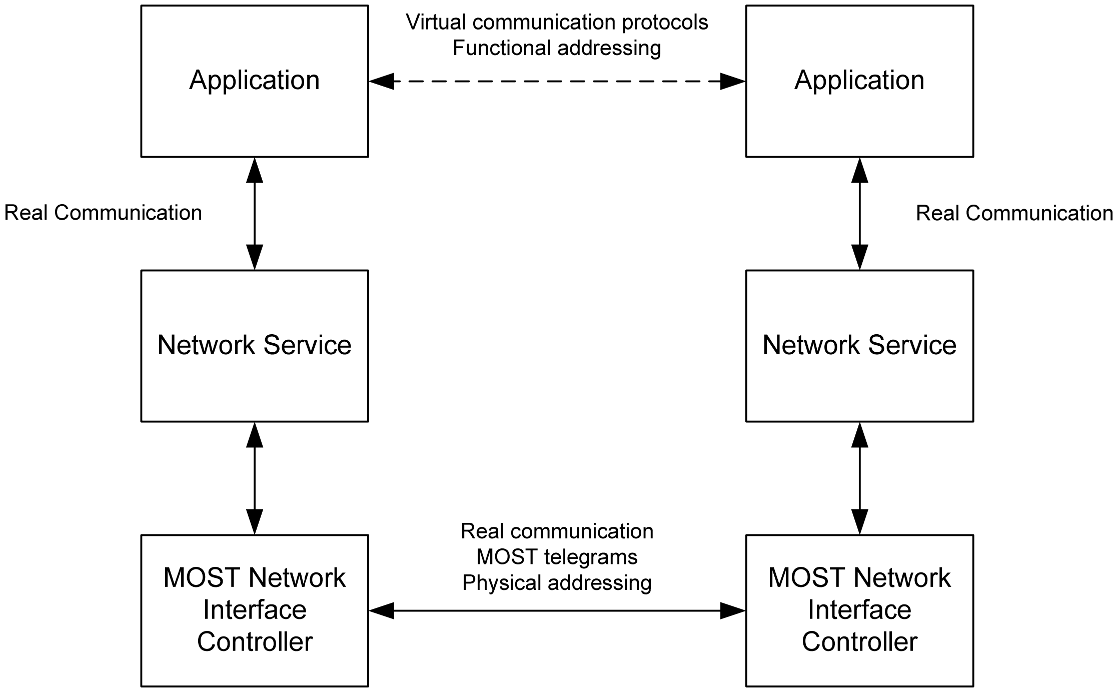


Figure 2-13: Virtual communication between two devices on application layer and real comm. via network

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

Here, the entire data flow of an interaction between two devices via the network layer is described. One device controls the functions of the other. The figure below shows the properties of an FBlock with the FBlockID CD and the InstID 1. The FBlock is found in device CD Player with the physical MOST address CDC.

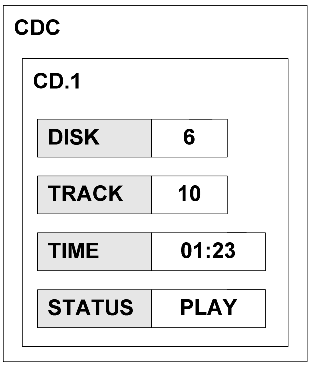


Figure 2-14: Device with MOST address CDC, an FBlock CD Player with FBlockID CD, and its functions

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 device 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 device address:

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

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

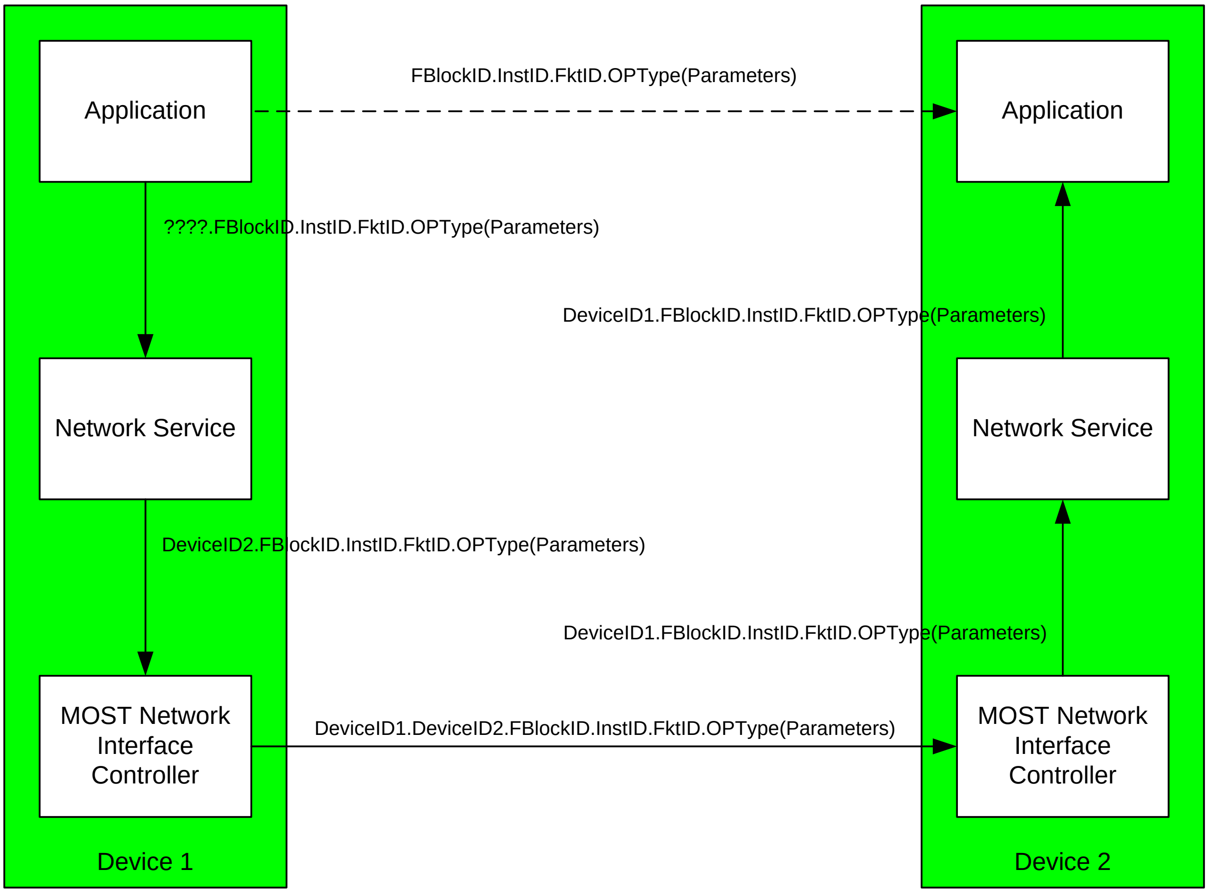

Application
FBlockID.InstID.FktID.OPType(Parameters)
Application
????.FBlockID.InstID.FktID.OPType(Parameters)
DeviceID1.FBlockID.InstID.FktID.OPType(Parameters)
Network Service
Network Service
DeviceID2.FBlockID.InstID.FktID.OPType(Parameters)
DeviceID1.FBlockID.InstID.FktID.OPType(Parameters)
MOST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

Figure 2-15: 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 FBlock Shadows for controlling the CD changer and Tuner FBlocks
- Controlled device (Slave):
  Contains only the CD changer FBlock

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

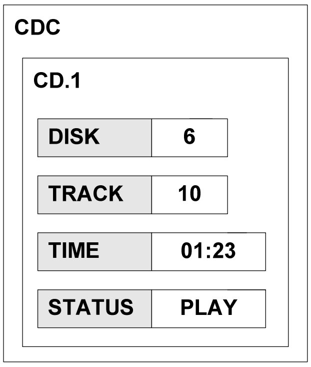


Figure 2-16: Example for a Slave device

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, for example, when the player changes to the next track during play mode (on the condition that another device is registered in the Notification Matrix of FBlock 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)
```

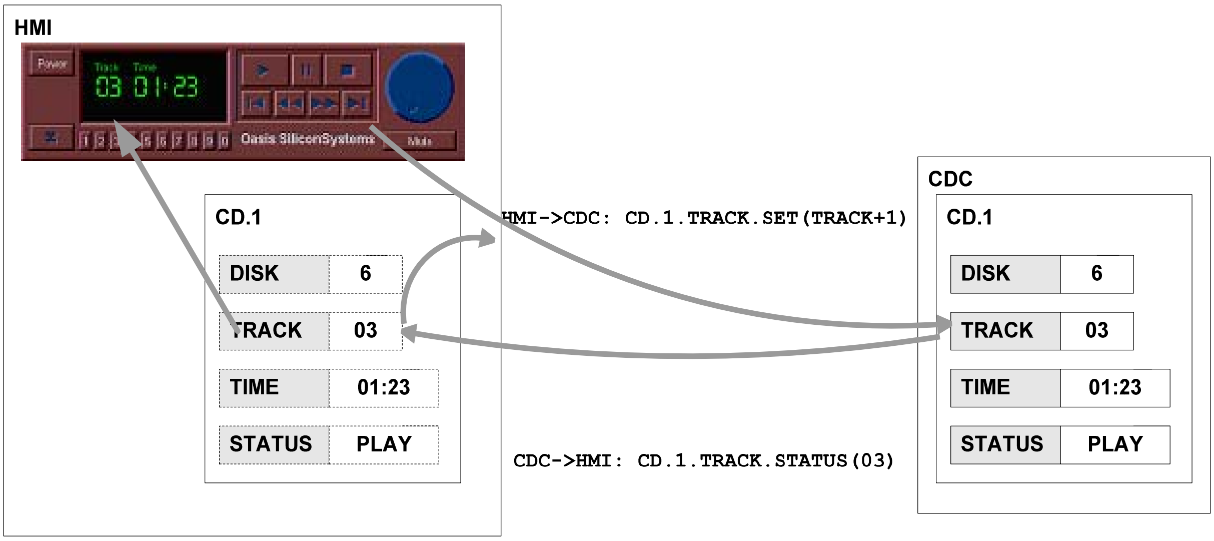
By this, two CD FBlocks can be addressed unambiguously, even if they are located within one physical device with one MOST address. This also guarantees that status reports can be assigned unambiguously:

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

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

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

The controlling device (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:



```
HMI->CDC: CD.1.TRACK.SET(TRACK+1)
CDC->HMI: CD.1.TRACK.STATUS(03)
```

Figure 2-17: Virtual illustration of the controlled properties in the control device

The HMI shown in Figure 2-17 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.

When the button is clicked, 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 FBlock. 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.

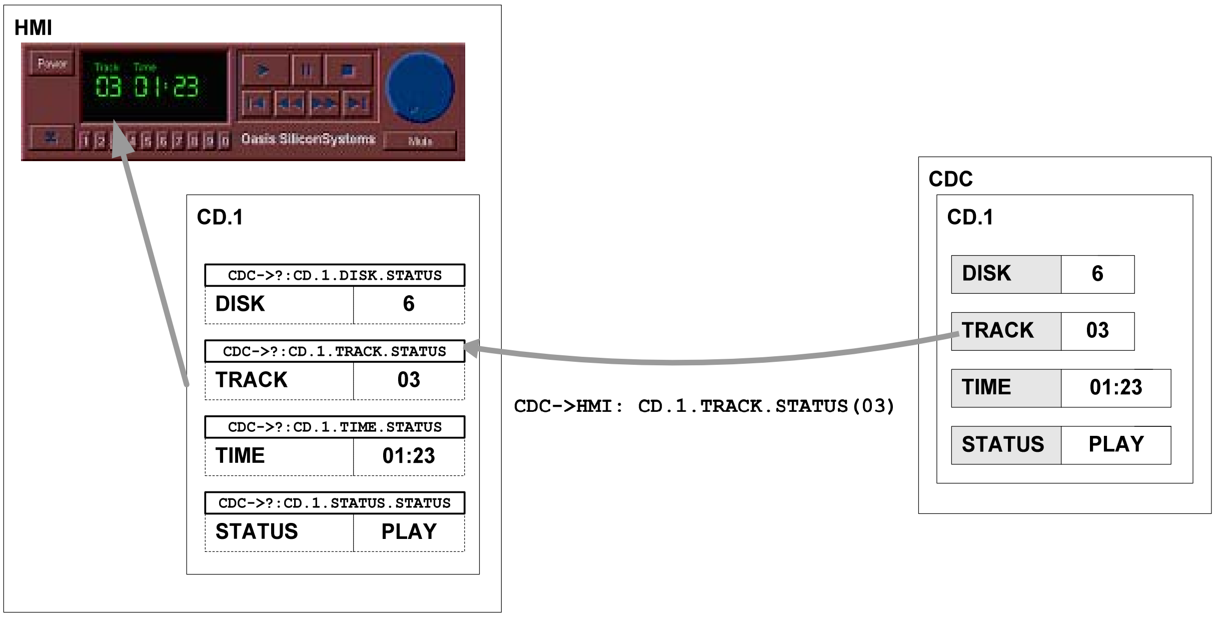


Figure 2-18: Unambiguous assignment between protocol and variable

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.

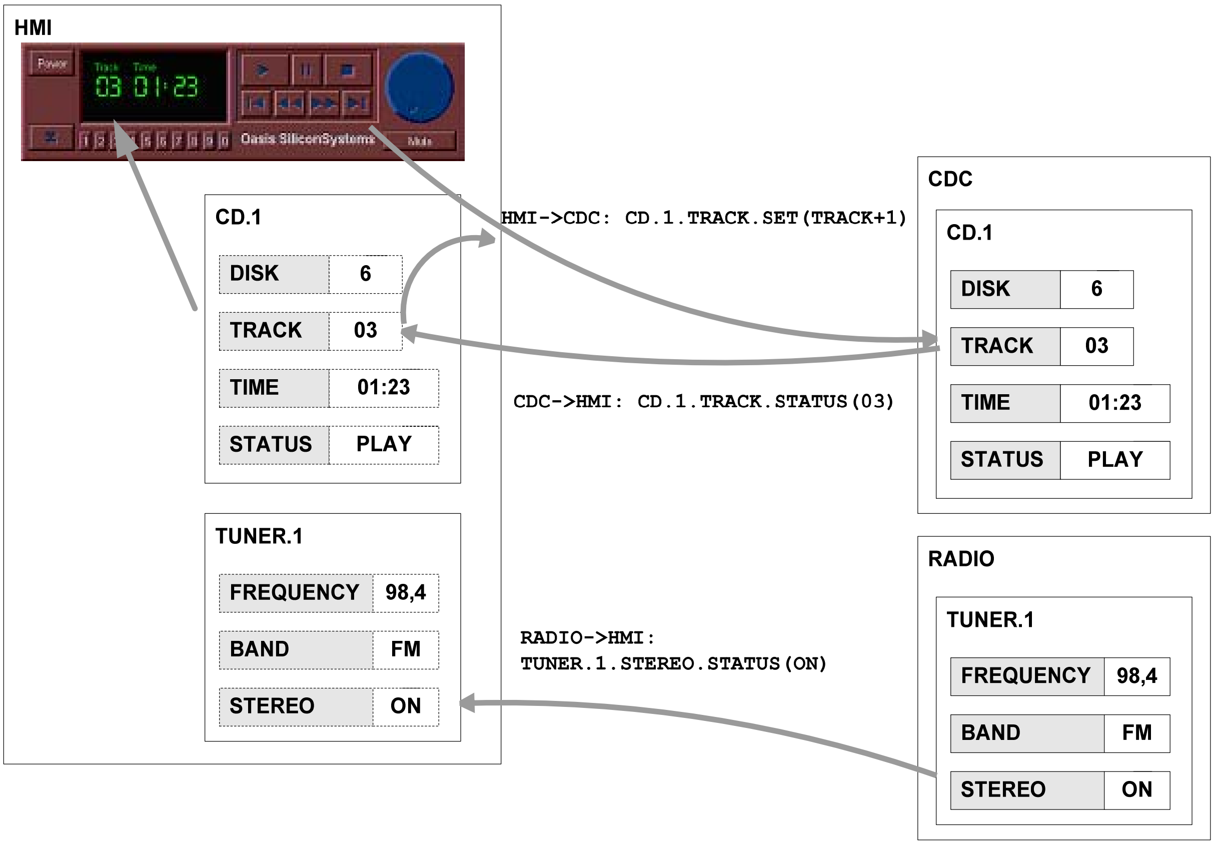


Figure 2-19: Controlling multiple devices

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 FBlocks are located in one device, handling would be identical.

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

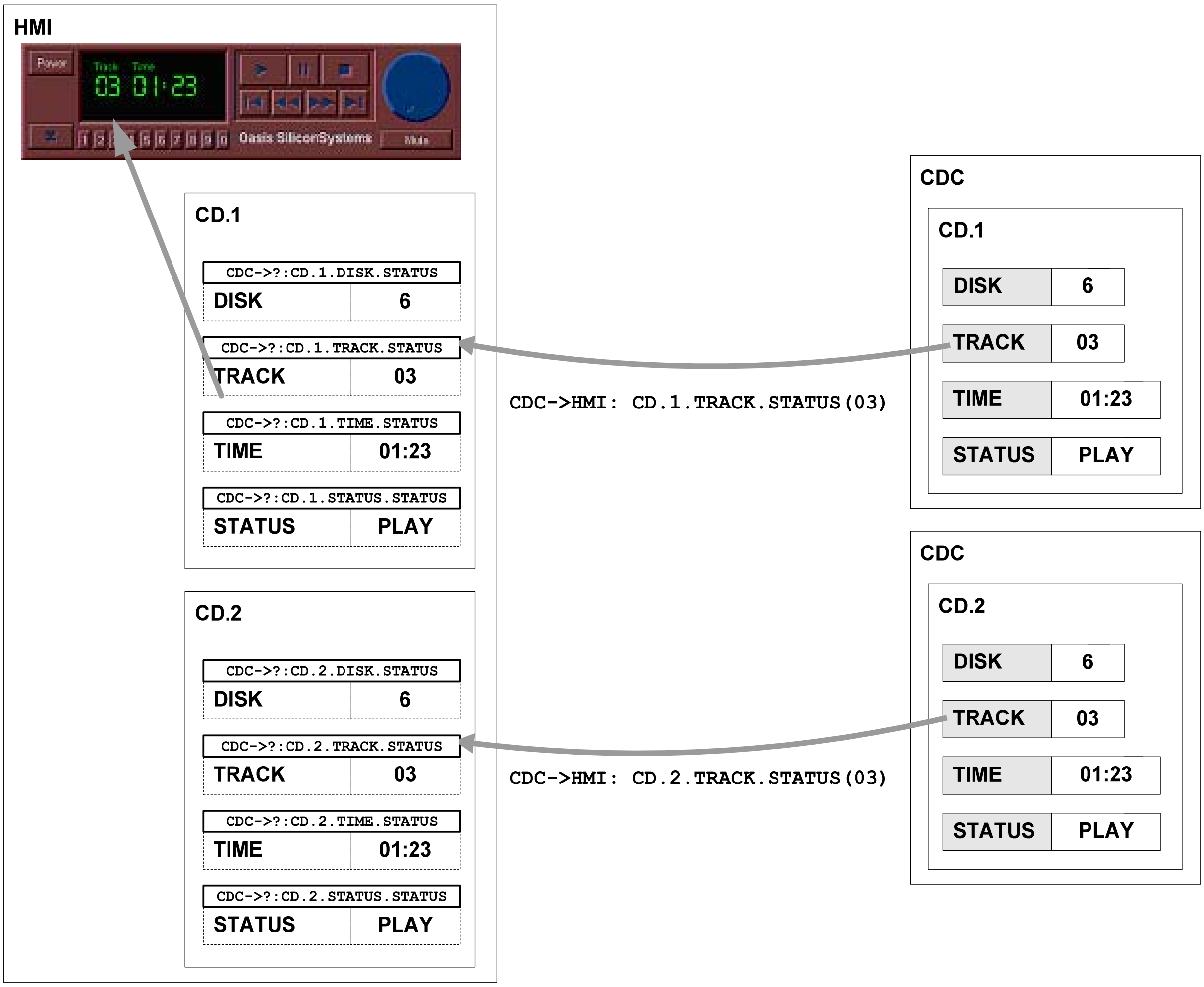


Figure 2-20: Controlling two identical devices

For the assignment of protocols and variables in the Controller, 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 that 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. This allows functions to 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 Controllers, 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.

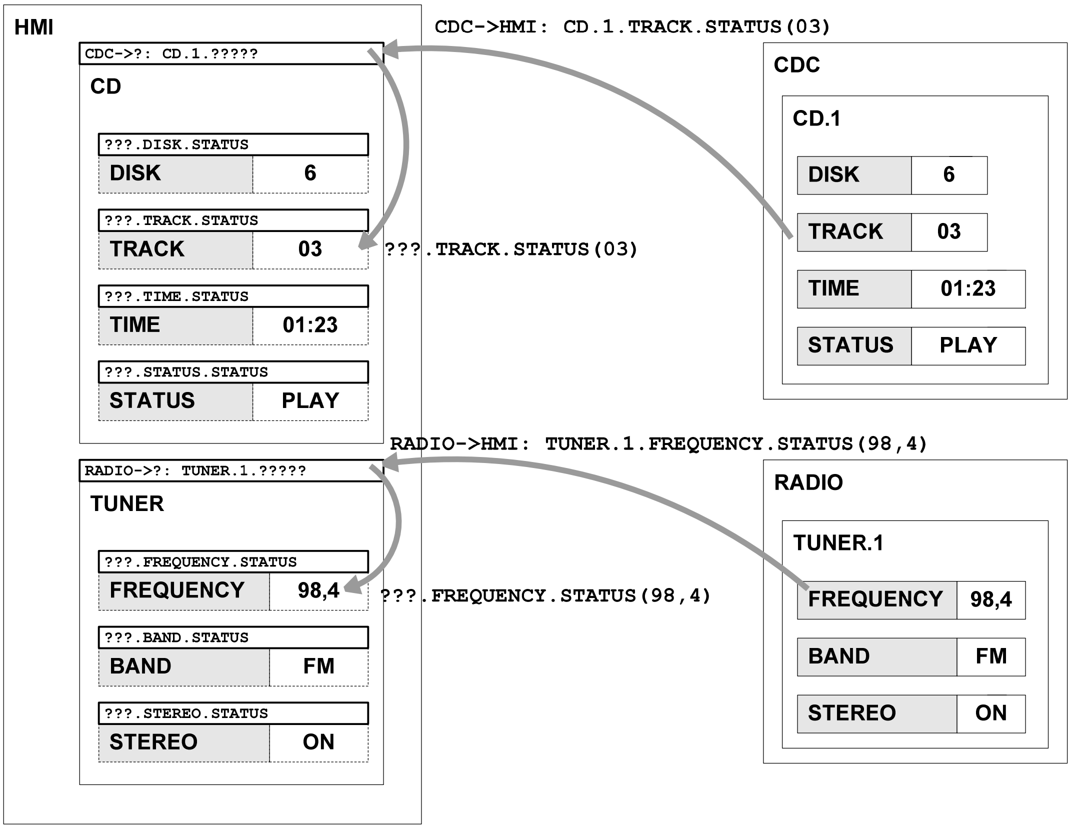


Figure 2-21: Hierarchical structure of the protocol filter (command interpreter)

Without object-oriented programming, the stepwise filtering can be implemented by using a message dispatcher. This dispatcher would forward the protocols to the respective FBlocks based on sender address, FBlockID and InstID. Every FBlock 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: 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 that 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.

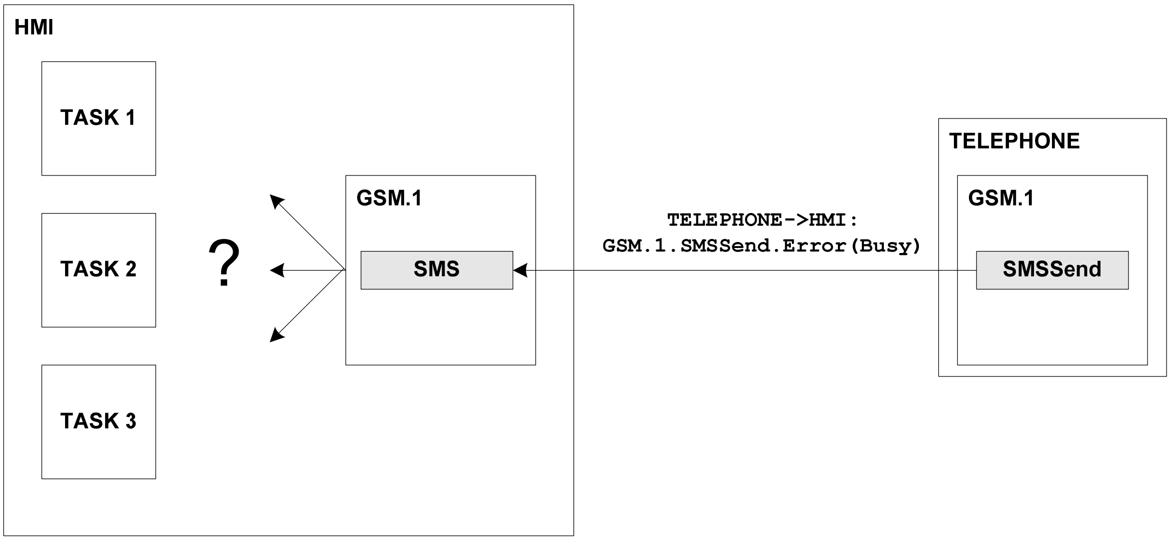


Figure 2-22: Routing answers in case of multiple tasks (in one Controller) using one function

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, an FBlock. This may happen in a self-configuring audio system with four or six active speakers. The audio Controller knows that FBlocks 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 an FBlock, 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 (
                RxTxLog.FBlockID.InstID,
                RxTxLog.FBlockID.InstID,...)
```

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

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

If the respective FBlock 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 FBlocks contained in 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 FBlock that most characterizes the device (e.g., Tuner in a radio device) is listed first. The NetBlock and FBlock EnhancedTestability do not need to be listed, as they are mandatory FBlocks in every device:

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

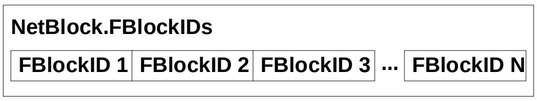


Figure 2-23: Reading the FBlocks of a device from NetBlock

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 an FBlock (e.g., simple or high-end audio amplifier). Therefore, every FBlock has the function **FktIDs** (0x000). It is read as follows:

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

Within an FBlock, FktIDs between 0x000 and 0xFFF (4096 different FktIDs) can be available. The FktIDs are assigned as described in 2.3.2 on page 28. This raises the problem of a compact response, if the functions contained in an FBlock are requested. It is solved by a mechanism derived from the Run Length Encoding (RLE). A bit field is built where the first bit is set to 1 if FktID 0x000 is available; the second bit is set to 1 if FktID 0x001 is available, and so on. Such a bit field may look like:

FktID		000	001	002	003	004	005	006	...	021	022	023	024	...	A00	A01	A02	A03	...	FFF
Bit field	1	1	1	0	0	1	1	0	0	0	1	1	0	0	1	1	0	0	0	0

The answer lists only the positions (FktIDs) where the bit state changes, beginning with an initial bit state of 1.

For the example shown above, the result would be:

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

The last 0 represents a stuffing nibble.

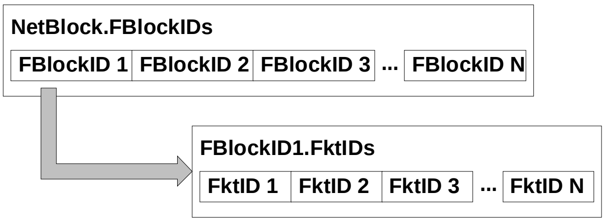


Figure 2-24: Requesting the functions contained in an application block

2.3.10 Transmitting the Function Interface

2.3.10.1 Principle

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

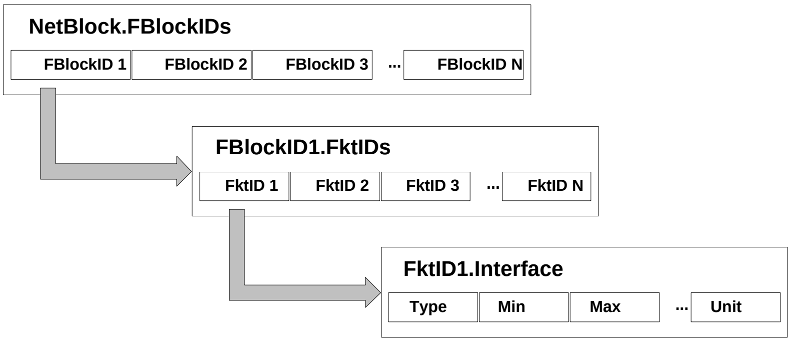


Figure 2-25: Requesting the function interface of a function

The flow for determining all function interfaces of an FBlock looks like:

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

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

2.3.10.2 Realization of the Ability to Extract the Function Interface

In the FBlock Specifications, every interface of classified functions is described. Thus, a classified definition of application protocols, as well as a uniform description is possible, which can be based onto a few classes, 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.

In the next sections, the following universal parameters are used:

Flags: 8 bits

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

By using the Visible bit, the device can influence whether the function is displayed at the moment or not (default = 1 = visible). It is possible to disable a function temporarily (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 2-7 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 2-7: 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 Packet Data 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 Packet Data 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 Packet Data 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, that is, over the Packet Data Channel.

Bits 6 and 7 are reserved for future use.

Class:	8 bits	0x00	Unclassified Method	
		0x01	Trigger Method	
		0x02	Sequence Method	
		0x10	Unclassified Property	
		0x11	Switch	
		0x12	Number	
		0x13	Text	
		0x14	Enumeration	
		0x15	Array	(Refer to section 2.3.11.2.2 on page 80.)
		0x16	Record	(Refer to section 2.3.11.2.1 on page 78.)
		0x17	Dynamic Array	(Refer to section 2.3.11.2.3 on page 83.)
		0x18	Long Array	(Refer to section 2.3.11.2.4 on page 85.)
		0x19	BoolField	
		0x1A	BitSet	
		0x1B	Container	
		0x1C	Sequence Property	
		0x1D	Map	
		0xFF	Abort	(No further specifications behind this location)

OPTypes: 16 bits

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 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 46) 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 2-8: Classes of functions with a single parameter

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

2.3.11.1.1 Function Class Switch

OPType	Parameters
Set	**Boolean**[1]
Get	
SetGet	Boolean
GetInterface	
Status	Boolean
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, no Unicode, notification**	**0000 1011 = 0x0B**
Class:	**Switch**	**0x11**
OPTypes:	**Get, SetGet, Status GetInterface, Interface, Error**	**1101 0000 0010 0110 = 0xD026**
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.

---

[1] In the next sections that introduce individual function classes, some parameters appear in **boldface** type inside the OPType tables. These marked parameters are described in detail below the tables.

2.3.11.1.2 Function Class Number

OPType	Parameters
Set	**Number**
Get	
SetGet	Number
Increment	**NSteps**
Decrement	NSteps
GetInterface	
Status	Number
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 as the Number parameter.
Units:	Uns. Byte	Unit

Unit	Encoding
none	0x00
Distance:	
cm	0x01
m	0x02
km	0x03
miles	0x04
Time:	
µs (Micro second)	0x10
ms (Millisecond)	0x11
s (Second)	0x12
min (Minute)	0x13
h (Hour)	0x14
d (day)	0x15
mon (Month)	0x16
a (Year)	0x17
Frequency:	
1/min	0x20
Hz	0x21
kHz	0x22
MHz	0x23
Volume:	
l (Liter)	0x30
gal (UK)	0x31
gal (US)	0x32
ccm	0x33
Consumption:	
l/100km	0x40
miles/gal	0x41
km/l	0x42
Speed and Acceleration:	
km/h	0x50
Miles/h	0x51
m/s	0x52
cm/s	0x53
°/s	0x54
m/s^2	0x55
Temperature and Pressure:	
°C	0x60
F	0x61
K	0x62
bar	0x63
psi	0x64
Miscellaneous:	
dB	0x70
%	0x71
Voltage:	
mV	0x80
V	0x81
Current:	
mA	0x90
A	0x91
Angle:	
Degrees	0xA0
Minutes	0xA1
Seconds	0xA2
$360°/ 2^{32}$	0xA3
$360°/ 2^{8}$	0xA4
Resolution:	
Pixel	0xB0
Data:	
Byte	0xC0
kByte	0xC1
MByte	0xC2

Table 2-9: Available units

2.3.11.1.3 Function Class Text

OPType	Parameters
Set	String
Get	
SetGet	String
GetInterface	
Status	String
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	
SetGet	Pos
Increment	NSteps
Decrement	NSteps
GetInterface	
Status	Pos
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	
SetGet	Content
GetInterface	
Status	Content
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
DataType:	Uns. Byte	Type of variable: 0x00 Unsigned Byte 0x02 Unsigned Word 0x04 Unsigned Long
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).

BYTE 1

D 7	D 6	D 5	D 4	D 3	D 2	D 1	D 0
					F. 10	F. 9	F. 8

BYTE 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	Parameters
Set	**SetOfBits**
Get	
SetGet	SetOfBits
GetInterface	
Status	SetOfBits
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	Parameters
Set	Classified Stream
Get	
SetGet	Classified Stream
GetInterface	
Status	Classified Stream
Interface	Flags, Class, OPTypes, Name, **MaxLength**
Error	ErrorCode, ErrorInfo

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

MaxLength:	Unsigned Word	MaxLength indicates the maximum 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
SetGet	Position, Data
Increment	Position, NSteps
Decrement	Position, NSteps
GetInterface	
Status	Position, Data
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. For parameter OPTypes only Set, Get, SetGet, 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**
Container	Flags, Class, OPTypes, Name, **MaxLength**

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.

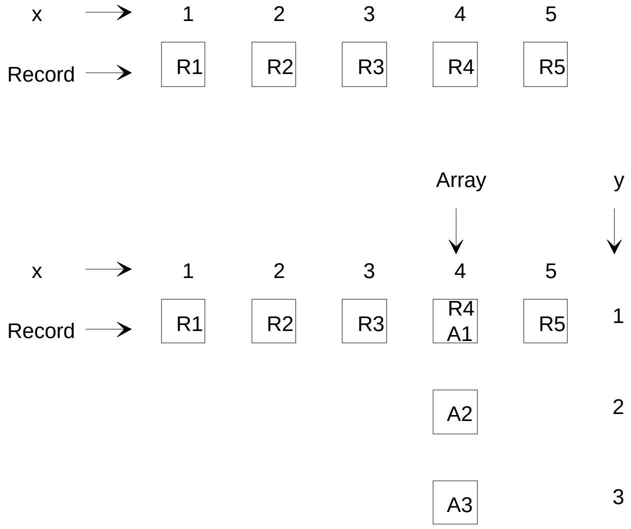


Figure 2-26: Meaning of position x in record (above) and of position y in a record with array (below)

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

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

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

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

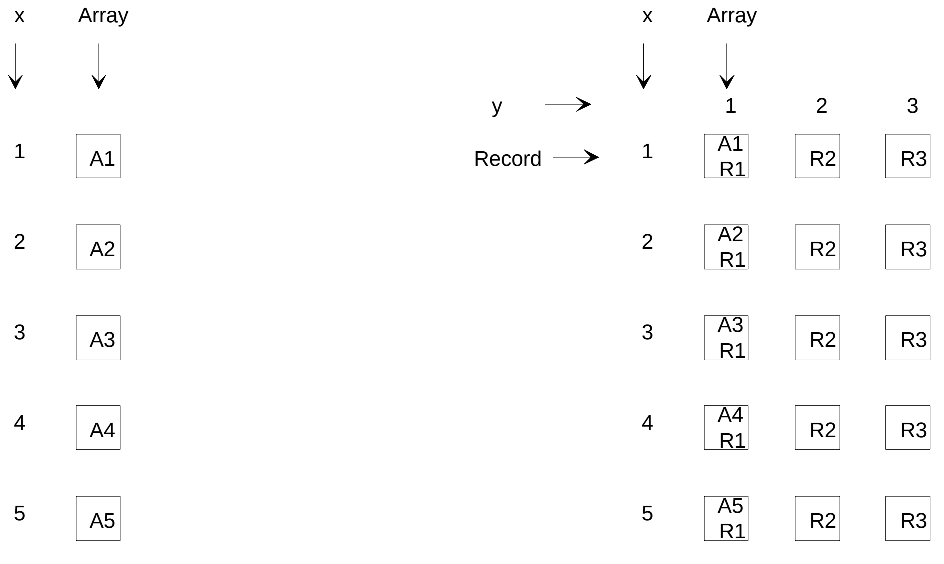


Figure 2-27: Position x in case of an array of basic type (left), y in case of an array of record (right)

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 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,for example, in case of station lists the pointer may point onto the PI of the station currently active. In other cases, the position may be more effective. This way can be very effective, if a single line shall be selected in several Arrays (e.g., an entry in all telephone directories).

2.3.11.2.3 Function Class DynamicArray

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

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

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

OPType	Parameter
Set	**Tag**, **PosY**, Data
Get	Tag, PosY
SetGet	Tag, PosY, Data
Increment	Tag, PosY, NSteps
Decrement	Tag, PosY, NSteps
GetInterface	
Status	Tag, PosY, Data
Interface	Refer to section 2.3.11.2.2 on page 80
Error	ErrorCode, ErrorInfo

Tag	Uns. Word	=	0x0000	all lines
		<>	0x0000	one special line
PosY	Uns. Byte	<>	0x00	one special column (only if Tag <> 0x0000)
		=	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 DynamicArray indicates the end. It starts with Tag 0xFFFF and contains dummy data. This line is included within the NMax counter.

<u>Examples for positioning:</u>

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

(1)

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

(2)

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

(3)

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

(4)

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

Editing In DynamicArrays:

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

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

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

Examples:

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

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

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

Examples:

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

Notification on DynamicArrays

In case of a notification update, always the complete DynamicArray is transmitted to the notified Controllers.

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.

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 MotherArray starts with Tag 0xFFFF and contains dummy data. The tag 0x000 is reserved to the Controller for initialization of the ArrayWindow. Therefore it must not be used in the MotherArray.

2.3.11.2.4.1 MotherArray

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

The main difference compared to other arrays is that the MotherArray is not controlled and viewed directly but via one or more different functions. 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.
There must be an individual ArrayWindow for each Controller. It is possible that several HMIs have individual ArrayWindows to an address directory (MotherArray), which have different size and position. Functions of class ArrayWindow are instantiated dynamically at runtime. Therefore an FBlock (that has a MotherArray, which shall be accessed by ArrayWindows) must provide a method CreateArrayWindow for instantiation and a method DestroyArrayWindow (both of class Unclassified Method). The FktIDArrayWindow is used as instance handle, which is transferred from Slave to Controller during instantiation.

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

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, for example, in a telephone all telephone directories are available as MotherArrays, every HMI that is interested in a telephone directory may instantiate an ArrayWindow for the respective MotherArray. So it can be that, for example, three telephone directories may be watched by three ArrayWindows.

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

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

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

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

A CreateArrayWindow with a PositionTag too close to the bottom of the MotherArray has the same effect as MoveArrayWindow.Bottom: It creates an ArrayWindow that contains the last WindowSize number of valid elements plus the last element with tag 0xFFFF.
If a Controller tries to position the ArrayWindow on a non-existent PositionTag, the ArrayWindow is positioned on the next greater tag. If there is no greater tag available, the next lesser tag is used.

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

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

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

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

Set (9473, 03, B3)

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

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

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

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

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

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

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

Tag	Uns. Word	unique handle of a row (0xFFFF no valid value) after which the given number of lines is deleted
FktIDArrayWindow	Uns. Word	FktID of ArrayWindow
Quantity	Uns. Word	number of rows

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

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

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

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
	SetGet	Tag, PosY, Data
	Increment	Tag, PosY, Nsteps
	Decrement	Tag, PosY, Nsteps
	GetInterface	
	Status	Tag, PosY, **CurrentSize**, **AbsolutPosition**, Data
	Interface	Refer to section 2.3.11.2.2 on page 80
	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 The termination line with Tag 0xFFFF is not counted for CurrentSize. CurrentSize only indicates the number of meaningful lines		
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 MoveArrayWindow is mandatory. An instance of MoveArrayWindow is used for all instances of ArrayWindows (FktID) of an FBlock.

`MoveArrayWindow.StartResultAck ( Senderhandle, FktIDMotherArray, MovingMode, Number, Tag )`

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

Top and Bottom:

Top and Bottom move the ArrayWindow to the start or the end of the MotherArray. The parameters Number and Tag are transferred as well but they are not used in this mode.
If an ArrayWindow is positioned to Bottom, it contains the last WindowSize valid elements plus the last element with tag 0xFFFF. This means that in this case the ArrayWindow is one element bigger than in other cases.

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

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

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

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

MotherArray

ArrayWindow

MoveArrayWindwow.StartResultAck (SenderHandle, FktID, Top, xx, xxxx)

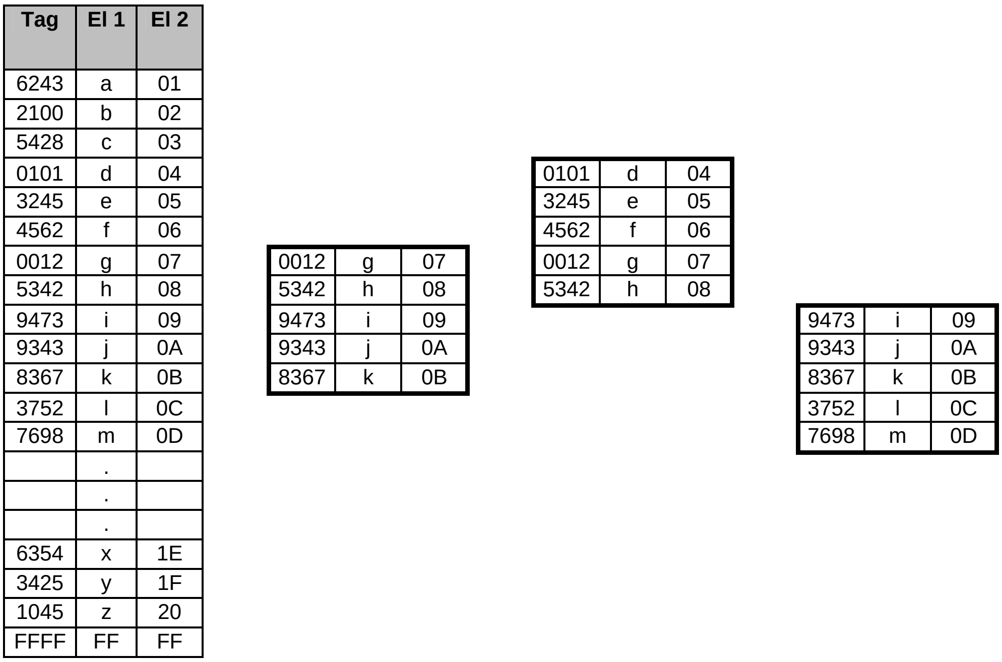
MoveArrayWindwow.StartResultAck (SenderHandle, FktID, Bottom, xx, xxxx)

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.
The ArrayWindow stays at the Top or Bottom position.

Please note:

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



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

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

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

MotherArray

ArrayWindow

MoveArrayWindwow.StartResultAck (SenderHandle, FktID, Up, 03, xxxx)

MoveArrayWindwow.StartResultAck (SenderHandle, FktID, Down, 05, xxxx)

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

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

MotherArray

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

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

ArrayWindow

MoveArrayWindwow.StartResultAck (SenderHandle, FktID, Absolute, xx, 2100)

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

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

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

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

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

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

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

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

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

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 exist. It works like a normal array except that it is only possible to perform a Get operation on it.

2.3.11.2.5 Function Class Map

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

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

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

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

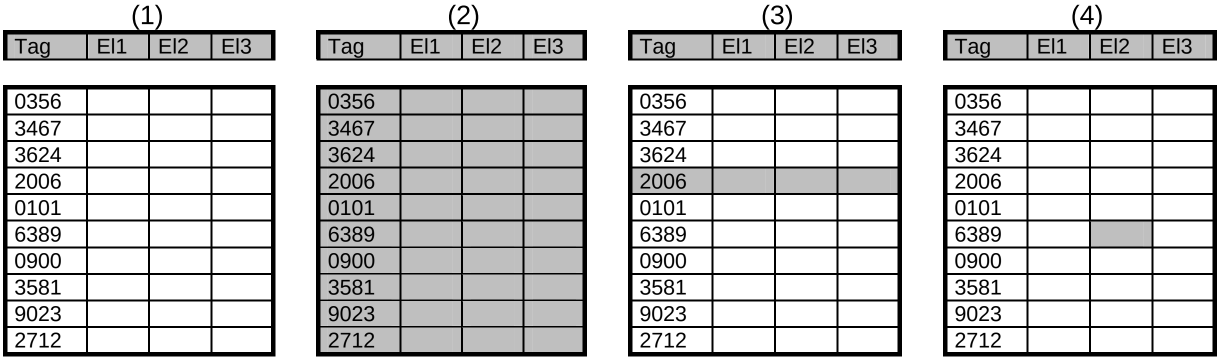
OPType	Parameter
Set	**Tag**, **PosY**, Data
Get	Tag, PosY
SetGet	Tag, PosY, Data
Increment	Tag, PosY, NSteps
Decrement	Tag, PosY, NSteps
GetInterface	
Status	Tag, PosY, {Data}
Interface	Refer to section **2.3.11.2.2** on page 80
Error	ErrorCode, ErrorInfo

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

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

Examples for Accessing Elements of a Map:

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



(1)

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

(2)

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

(3)

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

(4)

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

Status Transmission for Function Class Map:

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

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

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

1. Transmission of single entries:

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

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

2. Transmission of single lines:

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

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

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

3. Combined transmission of changes and insertions:

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

4. Transmission of whole Map property:

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

5. Removal of line:

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

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

Example 1: Notification of added line[1]

old list

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

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

Tag	El1	El2	El3
3624	031	032	033

new list

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

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

old list

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

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

Tag	El1	El2	El3
3467	N01	N02	N03

new list

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

---

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

Example 3: Notification of multiple modifications

old list

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

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

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

new list

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

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

old list

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

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

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

new list

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

Example 5: Deletion of line

old list

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

notification:
Tag = 0x2006, PosY = 0x00

new list

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

<u>Editing in Function Class Map:</u>

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

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

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

---

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

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

Examples:

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

Inserts two lines into the specified Map property.

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

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

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

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

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

Examples:

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

Deletes the line with Tag 0x0101.

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

Deletes the lines with Tags 0x3581 and 0x2006.

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

Deletes the entire array.

2.3.11.2.6 Function Class Sequence Property

OPType	Parameters
Set	<Parameter>{, <Parameter>}
Get	
SetGet	<Parameter>{, <Parameter>}
GetInterface	
Status	<Parameter>{, <Parameter>}
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 in case of elements, parameter Flags is not available. For parameter OPTypes only Set, Get, SetGet, Status 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 class of methods is used to trigger something. They have no parameters.
Sequence Method	This class of methods has a number of parameters. An interface description can be generated according to section 2.3.11.3.2.
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	Parameters
Start	
StartResult	
GetInterface	
StartResultAck	SenderHandle
ErrorAck	SenderHandle, ErrorCode, ErrorInfo
ProcessingAck	SenderHandle
Processing	
Result	
ResultAck	SenderHandle
Interface	Flags, Class, OPTypes, Name
Error	ErrorCode, ErrorInfo

2.3.11.3.2 Function Class Sequence Method

OPType	**Parameters**
Start	<Parameter>{, <Parameter>}
Abort	
StartResult	<Parameter>{, <Parameter>}
GetInterface	
StartResultAck	SenderHandle, <Parameter>{, <Parameter>}
AbortAck	SenderHandle
ErrorAck	SenderHandle, ErrorCode, ErrorInfo
ProcessingAck	SenderHandle
Processing	
Result	<Parameter>{, <Parameter>}
ResultAck	SenderHandle, <Parameter>{, <Parameter>}
Interface	Flags, Class, Name, **NElements, IntDesc1, InDesc2, …**
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 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 FBlocks. 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 FBlock. 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 FBlock, on the number of properties, and on the number of device entries, each of which must be registered individually.

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

- The Notification Matrix is only a model. It does not dictate the software implementation method.
- Implementation may be done in very economical ways, for example, 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 FBlocks, 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 53, 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 device registers for a property that has already been registered for this device, the report is sent as if the device had been registered for the first time. However, this does not generate a second entry in the Notification Matrix. This also applies to registering with group addresses.

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

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

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

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

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

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

Please note:
In case of array properties, only those elements that have been changed are sent as status during notification. This applies to regular arrays. The content of DynamicArrays and ArrayWindows is always sent entirely.

If a device sends a status message based on a notification and the MOST Network Interface Controller indicates to the Network Services that the transmission failed due to an invalid target address, all entries for this target address shall be deleted from the Notification Matrix of the FBlock sending this status message.

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

Error handling:

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

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

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

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

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

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

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

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

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

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

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

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

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

  Application Example:
  This error code could be used to indicate the failure of a sensor. If the sensor signal disappears, the Sensor function would report the error “Not available” (0x41). If the function has an implemented notification mechanism, the error is distributed to the registered Controllers.
  This message is also sent, in case a node registers for the property after the problem occurred. Failure of a whole FBlock is described in section 3.2.5.4.

---

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

Reactions on system events

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

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

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

3 Network Section

3.1 MOST Network Interface Controller and its Internal Services

The MOST Network Interface Controller provides the means for operating the MOST bus simply and safely, and for the transmission of different types of data. Based on these mechanisms, higher layers are defined. The following sections give an overview of the features of the MOST Network Interface Controller that are available for the use in higher layers.

3.1.1 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.

oPhy
When using oPhy, the bypass is closed, while the MOST Network Interface Controller is reset.

ePhy
During reset of the MOST Network Interface Controller using ePhy, the output is disabled.
This is caused by the ePhy coding. During this time all devices located downstream detect unlocks. The bypass state is first available after the MOST Network Interface Controller is coming out of reset.

3.1.2 Master/Slave

A MOST system consists of up to 64 nodes. By configuration, any of the MOST Network Interface can be the TimingMaster; all the others are Slaves. The TimingMaster provides generation and transporting of system clock, the frames, and blocks. All Slave devices derive their clock from the MOST bus.

3.1.3 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 recognition (3.5.2.3 Compensating Network Delay), and Packet Data Channel management.

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

The sample frequency in a MOST system can be chosen in a range between 30 kHz and 50 kHz. 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.

3.1.3.1 Frames

The MOST frame structure is designed to allow for easy re-synchronization as well as clock and data recovery while guaranteeing data quality and integrity. Built-in structures provide the basis for simple network management on the lowest layers to avoid overhead.

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

MOST25
One MOST25 frame consists of 512 bits (64 bytes). The first byte is used for administrative purposes. The next 60 bytes are used for stream and packet data transfer, where the Boundary is defined in 4 byte steps. The Boundary can only have values between 6 and 15. Therefore, at least 24 bytes are available for Stream data transfer. All Stream data bytes are transmitted before any Packet data bytes occur. The next two bytes of each frame are reserved for Control data and the last byte is another administrative byte.

This diagram represents the MOST25 frame.

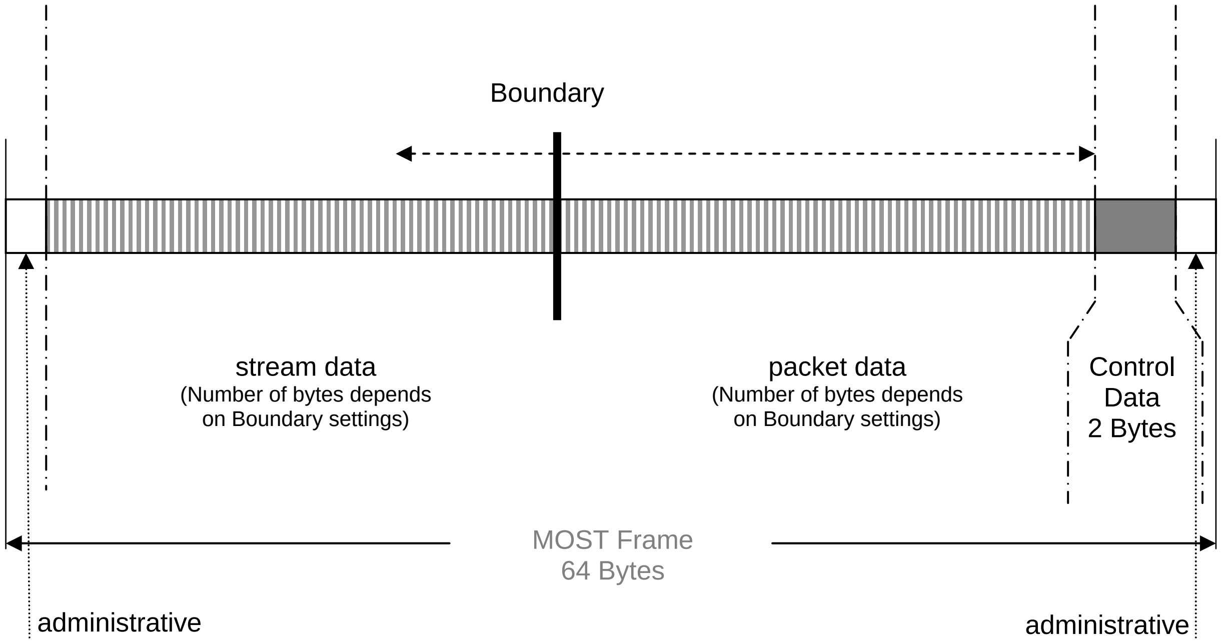


Figure 3-1: MOST25 frame

A MOST25 Control Message has a fixed size of 32 bytes and is time multiplexed over 16 MOST frames, each having two Control data bytes.

Byte Number	Task
0	Administrative ° Preamble (bits 0-3) ° 4 bits Boundary Descriptor (bits 4-7)
1 - 60	60 data bytes
61 - 62	2 data bytes for Control Messages
63	Administrative ° Frame control and status bits ° Parity bit (last bit)

Table 3-1: Structure of the MOST25 frame

MOST50
Due to the fact that MOST50 is designed for high bandwidth, one MOST50 frame consists of 1024 bits (128 bytes). The first 11 bytes are used for administrative purposes. In this area 4 bytes are used for Control data. Therefore, a MOST Control Message gets multiplexed into 4 bytes/frame pieces. The Control Message length can vary depending on the specifics of the particular Control Message. This results in better utilization of the bandwidth regarding Control Messages. The next 117 bytes are used for Packet and Stream data transfer.

This diagram represents a MOST50 frame.

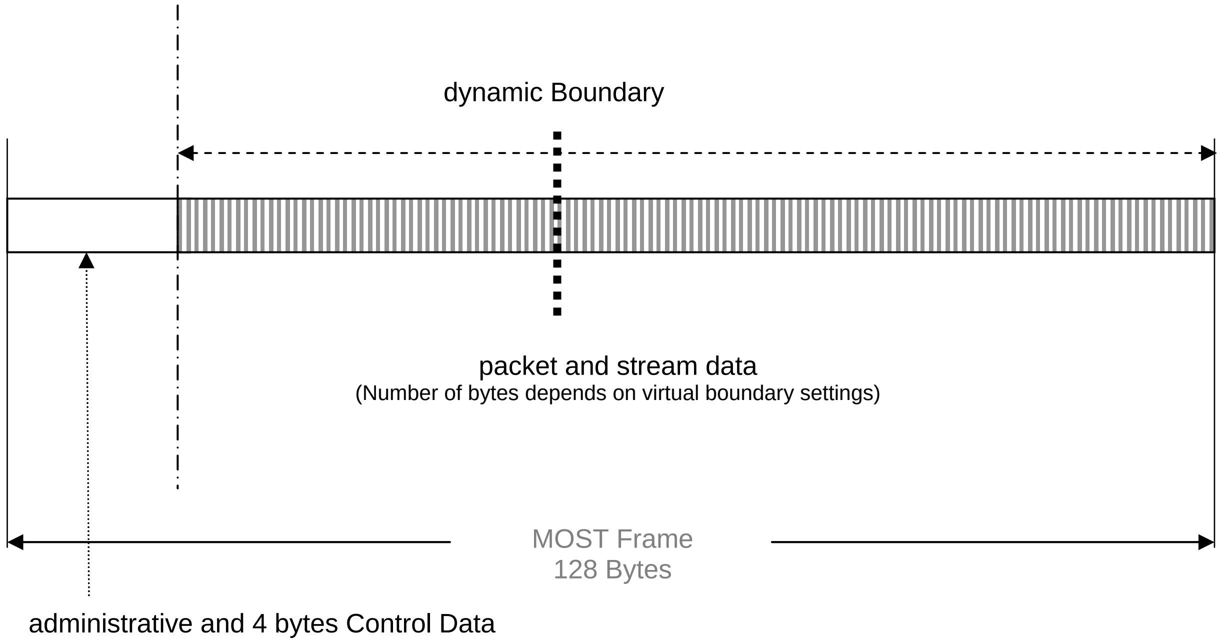

dynamic Boundary
packet and stream data
(Number of bytes depends on virtual boundary settings)
MOST Frame
128 Bytes
administrative and 4 bytes Control Data

Figure 3-2: MOST50 frame

Byte Number	Task
0 - 10	Administrative, includes 4 Control data bytes, additionally ° System Lock flag ° Boundary Descriptor
11 - 127	117 data bytes

Table 3-2: Structure of the MOST50 frame

3.1.3.1.1 Preamble

The preambles are used internally to synchronize the MOST core and its internal functions to the bit stream.

For synchronization to a frame, two different mechanisms are used for 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 to the Master node. In a Master node, the transmitted bit stream is synchronized to an external timing 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.

3.1.3.1.2 Boundary Descriptor

MOST25
The Boundary Descriptor provides a flexible way of changing the bandwidth for streaming and packet data transmission. It represents the number of 4 byte blocks (quadlets) of data used for streaming data. This value is used to determine the boundary between the streaming and packet data areas in the frame. A count value of zero indicates no streaming data and 15 quadlets of packet data, while a count value of 15 indicates 15 quadlets of streaming data and no packet data.

By this means, a 60 byte data field can be allocated to either streaming or packet 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 packet data bytes per frame is 36 bytes, which means that the Boundary Descriptor values can be between 6 and 15. When the Boundary Descriptor is set to the minimum value of 6, 24 bytes per frame remain for streaming data.

The Boundary Descriptor is managed by the TimingMaster of a MOST Network (see 3.7.1).

After having changed the Boundary Descriptor, the streaming connections must be re-built.

The Boundary can be seen as a real border between the area that is available for streaming data and the area that is used for packet transmission.

Bandwidth calculation:
Total bandwidth = 60
Packet bandwidth = 60 – Boundary * 4
Streaming bandwidth = Boundary * 4

MOST50
MOST50 systems have access to a Dynamic Boundary. This means that the bandwidth which is available for the streaming data transmission can be changed during runtime of the system, that is, during NetInterface state NetOn.

The initial value, which is used after each reset of the TimingMaster, is set in the TimingMaster's configuration. A system might not have the need to change the Boundary dynamically, but the Network Interface provides that possibility.

To change the Boundary during runtime the command NetBlock.Boundary.SetGet is sent to the NetBlock with instance ID zero (i.e., NetBlock in TimingMaster). A Slave device does not provide this property.

The only application that can allow or deny such a change is the application running on top of the TimingMaster.

MOST50 provides the possibility to use nearly the total bandwidth (116 out of 117 bytes per frame) for packet data transmission.

Bandwidth calculation:
The total bandwidth, that is, the sum of bandwidth for streaming and packet data, is 117 bytes per frame:

Total bandwidth = Packet bandwidth + Streaming bandwidth = 117

The bandwidth to be used for packet transmission in number of bytes per frame is calculated according to following formula:

Packet bandwidth = Total bandwidth – (Boundary * 4) – 1 = 116 – (Boundary * 4)

The remaining bandwidth can be used to transport data of type streaming:

Streaming bandwidth = (Boundary * 4) + 1

Packet/Streaming Data usage example:

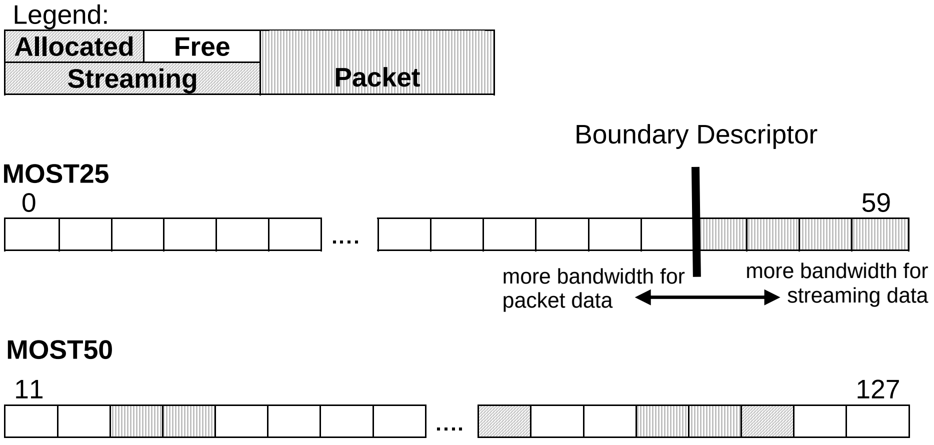


Figure 3-3: Packet/Streaming Data usage example

Property NetBlock.Boundary is designed to control the available bandwidth for the packet transmission channel (asynchronous data transmission). It is possible to use the total available bandwidth for streaming data.

NetBlock. Boundary	**MOST25**		**MOST50**	
	Available bandwidth for streaming data (in number of bytes per frame)		Available bandwidth for streaming data (in number of bytes per frame)	
0	N/a	N/a	1	116
1			5	112
2			9	108
3			13	104
4			17	100
5			21	96
6	24	36	25	92
7	28	32	29	88
8	32	28	33	84
9	36	24	37	80
10	40	20	41	76
11	44	16	45	72
12	48	12	49	68
13	52	8	53	64
14	56	4	57	60
15	60	0	61	56
16	N/a	N/a	65	52
17			69	48
…			…	…
28			113	4
29			117	0

Table 3-3: NetBlock.Boundary influence in MOST25 and MOST50 systems

3.1.3.1.3 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.

3.1.3.2 Control Data Transport

MOST25
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. 2 bytes in each frame transport control data. The 2 bytes of 16 frames (1 block) are added to the control frame that transports a control telegram.

MOST 50
Control Messages are based on a variable number of frames.

- The minimum number of frames (per Control Message telegram) is 6, in case of TelLen 0 (command without payload).
- The maximum number of frames is 9, in case of TelLen 12.

3.1.3.3 Source Data

3.1.3.3.1 Distinction between Source Data and Control Data

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

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

3.1.3.3.2 Differentiating Streaming and Packet Data

MOST25
60 data bytes (15 quadlets) total are available for streaming and packet data.

MOST50
117 data bytes are available for streaming and packet data.

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

3.1.3.3.3 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.

3.1.3.3.4 Transparent Channels

Another use case for Source Data transmission is the transparent transmission of a RS232 interface, that is, without synchronizing RS232 to the bus. For this purpose, the original data is oversampled (depending on the system's sample rate, and on the sampling rate chosen for the transparent port) and routed via the network.

3.1.3.3.5 Streaming Data

The Streaming Data 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.

MOST25
The maximum number of bytes available in a Streaming Data Channel is 60 bytes/frame, which corresponds 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.

MOST50
The maximum number of bytes available in a Streaming Data Channel is 117 bytes/frame, which corresponds to 117 x 8 bits or 29 stereo channels of CD-quality audio. The typical frame rate is 48,000 frames/second.
The functionality of the routing engine is encapsulated in the NIC. Accessing Streaming Channels is handled via an abstract socket mechanism.

3.1.3.3.6 Packet Data

Another time slot is available for packet data transport as required for burst-like data. In contrast to the control data channel, the Packet 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 the Packet Data 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 packet data 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 3-4: Structure of a frame of packet data (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 3-5: Structure of a packet data frame in the packet data area (alternative data link layer)

MOST50
Only Table 3-5 is valid for MOST50.

Since the packet data area is variable, it can take several frames to complete a message. The corresponding management such as arbitration and channel allocation is provided by the MOST Network Interface Controller. A hardware CRC is provided. The CRC is calculated in the background and can be indicated in a register at the end of each packet data message. A low-level retry mechanism is not implemented.

3.1.3.4 Control Data

3.1.3.4.1 Control Data Interface

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

3.1.3.4.2 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.

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 3-6: Structure of a control data frame

MOST25
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 provide control of applications.

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

MOST25
System messages handle system-related operations such as resource handling for streaming data connections.

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).

3.1.4 Internal Services

3.1.4.1 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.
- Logical node address (2 bytes).
  This address can be set by the application.
  A logical node address can be either dynamic or static. A dynamic address is calculated based on the node position address (see 3.4.1). A static address is predefined and is not recalculated upon a state transition to System State NotOk.
- 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…0x043F	Node position (0x0400 + POS) address range.
0x440...0x4FF	Reserved
0x0500…0x0FEF	Static address range
0x0FF0	Optional debug address
0x0FF1…0x0FFD	Reserved
0x0FFE	Init address of Network Service
0x0FFF	Init address of Network Interface Controller
0x1000…0xFFFE	Reserved for future use
0xFFFF	Uninitialized logical node address

Table 3-7: 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.

3.1.4.2 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 TimingMaster 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.

3.1.4.3 Automatic Allocation Mechanism

MOST25
Bandwidth administration is supported by the MOST system resource administration on the MOST Network Interface Controller level.

Allocating bandwidth is done via a request from an application to the TimingMaster of the network. Bandwidth can be allocated with a granularity of one byte. 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.

Detection of unused channels, that is, channels that are allocated by a device but no longer used, is done with the help of the channel resource allocation table. Only the TimingMaster 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.

MOST50
Bandwidth is allocated implicitly by creating a socket, where the size of the socket denotes the amount. By simply destroying the socket, the allocated bandwidth is de-allocated.
This approach relies on a connection label (16-bit word) combined with bandwidth information (also 16-bit word), instead of a long list of single byte channels.
Furthermore, the routing and the low-level allocation are entirely encapsulated by the Network Interface Controller, and of no influence on the application.

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 node, that is, the physical interface, the MOST Network Interface Controller, and the Network Service.

The following classes of nodes exist in MOST:

- Primary Nodes
  (implement NetBlock, FBlock ET and Network Services)

MOST25

- Secondary Nodes
  (Secondary Nodes are controlled by Primary Nodes. A Primary Node might control more than one Secondary Node)

A MOST device contains at least one Primary Node and may contain additional nodes of the above mentioned classes. If the device contains more than one node it is called a multi-node device.

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 physical interface, and the Controller on which the Network Service are running. The Network Service occupies 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.

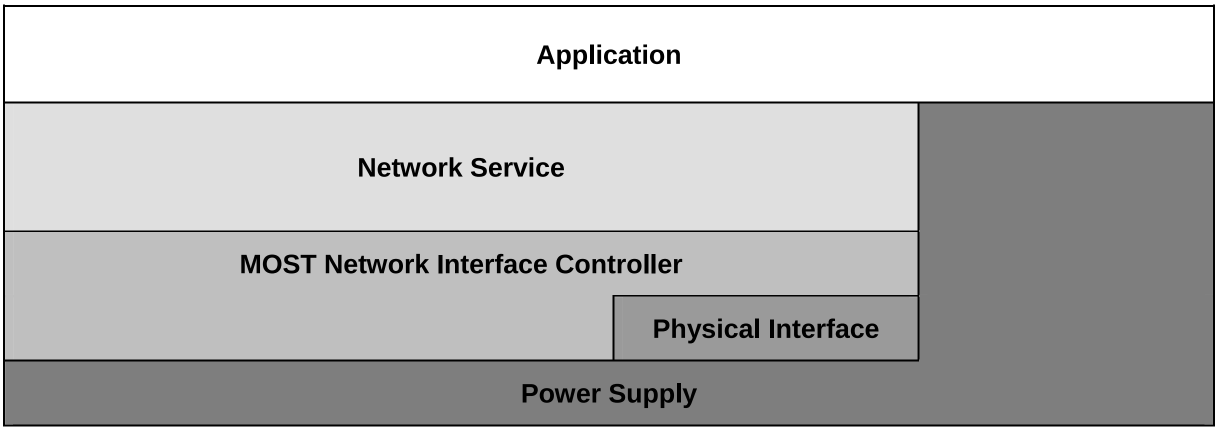


Figure 3-4: Layer model of a device

Generally, for each device, the device specification must define all the possible combinations of the states of the application section and the communication section. Particularly 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 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.
2. **DeviceNormalOperation:** The communication section, as well as the application, is in state normal operation.

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

DevicePowerOff:

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

DeviceStandBy:

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

DeviceNormalOperation:

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

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 202. 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.

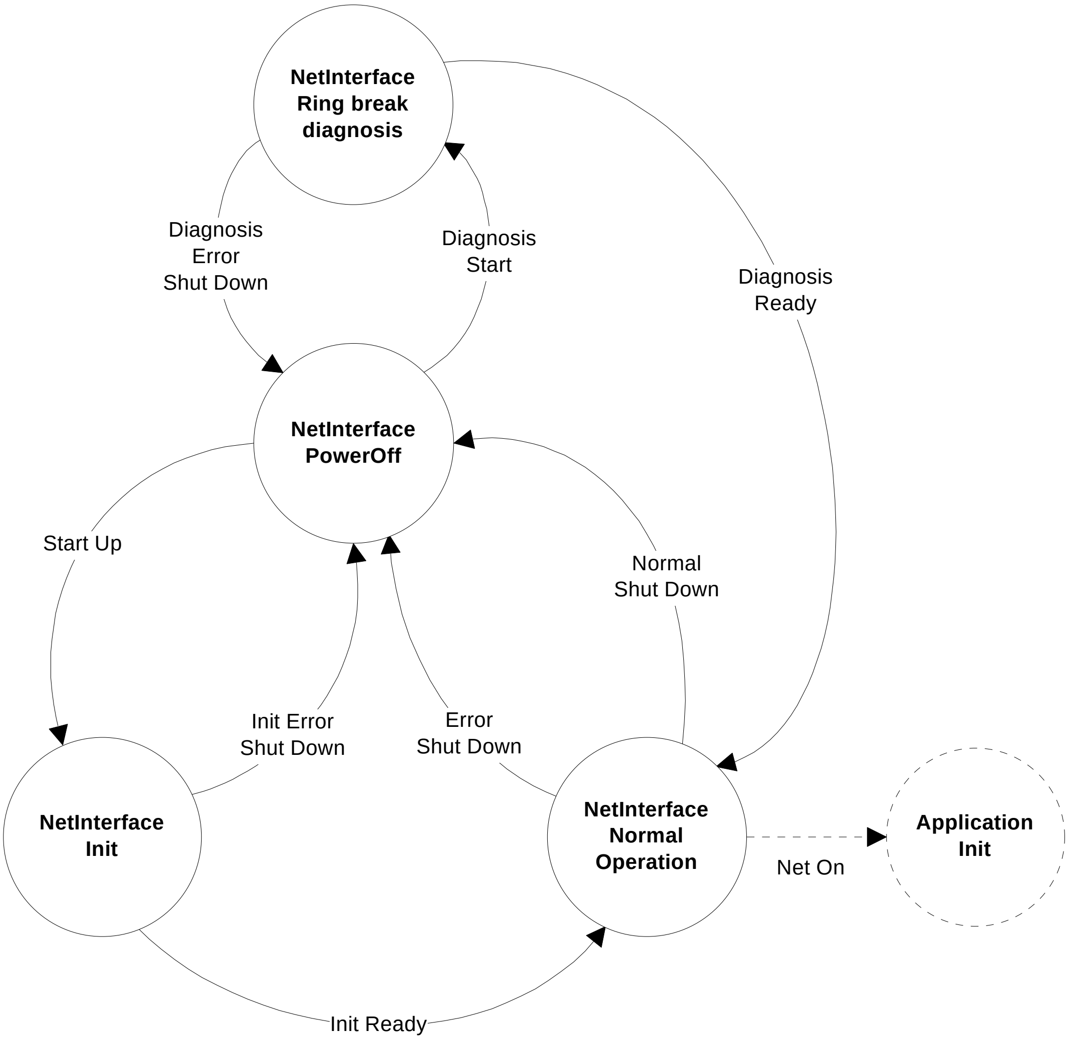


Figure 3-5: Flow chart “Overview of the states in NetInterface”

3.2.2.1 NetInterfacePowerOff

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

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

Event	Transition to	Cause
Start Up	NetInterfaceInit	A NetInterface is activated either by a modulated signal at the receiving physical interface, by the application (hypothetical example: phone receives a call), or by a switch at the device.
Diagnosis Start	NetInterfaceRingBreakDiagnosis	A NetInterface is activated by connecting to power (for information about signal SwitchToPower please refer to section 4.1 on page 202)

Table 3-8: 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.

MOST25
When entering state NetInterfaceInit, the TimingMaster loads the Boundary Descriptor with the “invalid” value 0x04. This value is transferred to all MOST Network Interface Controllers. As soon as the TimingMaster recognizes a stable lock, it sets the Boundary Descriptor to a valid value (>0x05).

MOST50
When entering state NetInterfaceInit, the TimingMaster clears the System Lock flag on data link layer. This value is transferred to all MOST Network Interface Controllers. As soon as the TimingMaster recognizes a stable lock, it sets the System Lock flag in the administrative area of the MOST frame.

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

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

Table 3-9: Events in state NetInterfaceInit

Causes for event Init Ready:

- In the Master device:
  Stable lock (for a minimum time of t_{Lock}) was recognized. Lock is called stable if for a period of time t_{Lock} no unlock events occurred.
- In a Slave device:
  Stable lock (for a minimum time of t_{Lock}) was recognized by the TimingMaster. Note that the approach is speed grade dependent.

MOST25 The Boundary Descriptor is set to a valid value (> 5).

MOST50 This is signaled on data link layer level via the System Lock flag.

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 a modulated signal was recognized, or a closed ring was recognized; or the modulated signal was switched off again.

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

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

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

Device with the TimingMaster

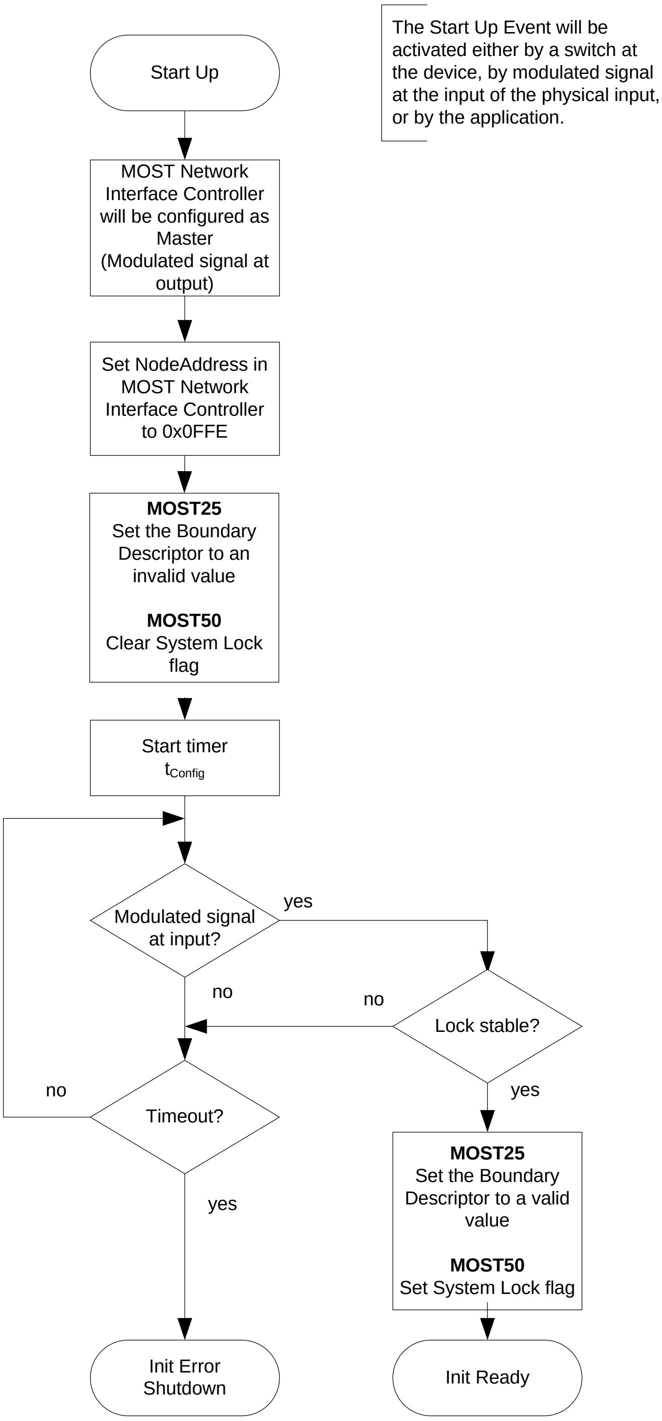

The Start Up Event will be activated either by a switch at the device, by modulated signal at the input of the physical input, or by the application.
Start Up
MOST Network Interface Controller will be configured as Master (Modulated signal at output)
Set NodeAddress in MOST Network Interface Controller to 0x0FFE
MOST25
Set the Boundary Descriptor to an invalid value
MOST50
Clear System Lock flag
Start timer t_{Config}
Modulated signal at input?
yes
no
no
Lock stable?
no
Timeout?
yes
yes
MOST25
Set the Boundary Descriptor to a valid value
MOST50
Set System Lock flag
Init Error Shutdown
Init Ready

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

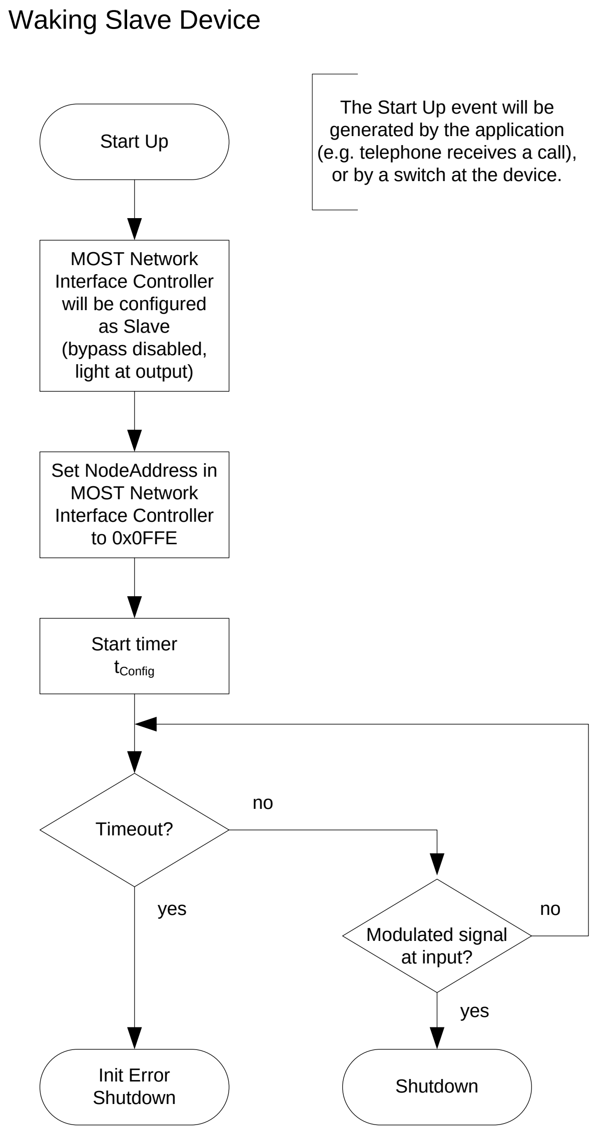

Waking Slave Device
Start Up
The Start Up event will be generated by the application (e.g. telephone receives a call), or by a switch at the device.
MOST Network Interface Controller will be configured as Slave (bypass disabled, light at output)
Set NodeAddress in MOST Network Interface Controller to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

Figure 3-7: Behavior of a waking Slave device in state NetInterfaceInit

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

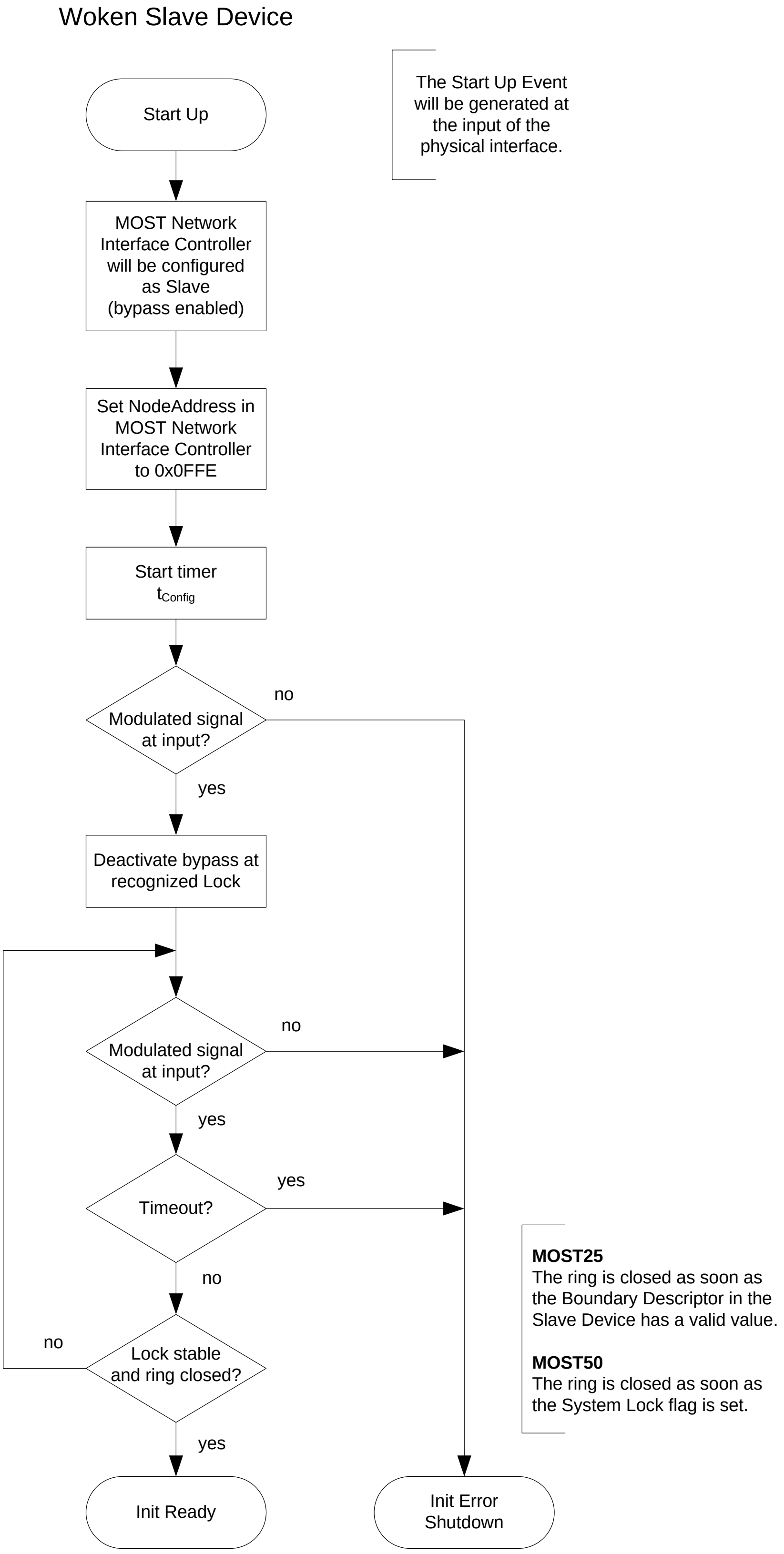

Woken Slave Device
Start Up
The Start Up Event will be generated at the input of the physical interface.
MOST Network Interface Controller will be configured as Slave (bypass enabled)
Set NodeAddress in MOST Network Interface Controller to 0x0FFE
Start timer t_{Config}
Modulated signal at input?
no
yes
Deactivate bypass at recognized Lock
Modulated signal at input?
no
yes
Timeout?
yes
no
Lock stable and ring closed?
no
yes
MOST25
The ring is closed as soon as the Boundary Descriptor in the Slave Device has a valid value.
MOST50
The ring is closed as soon as the System Lock flag is set.
Init Ready
Init Error Shutdown

Figure 3-8: 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 the Init Ready 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 the modulated signal.
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 3-10: Events in state NetInterfaceNormalOperation

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

In state NetInterfaceNormalOperation, the Network Service checks the lock state of the PLL of the MOST Network Interface Controller. If an unlock should occur, the application is informed as soon as possible by an unlock event (Error Shut Down). Please refer to section 3.2.5.2 ‘Unlock’ on page 147 for more information regarding this type of error.

The flow chart below shows the behavior in state NetInterfaceNormalOperation:

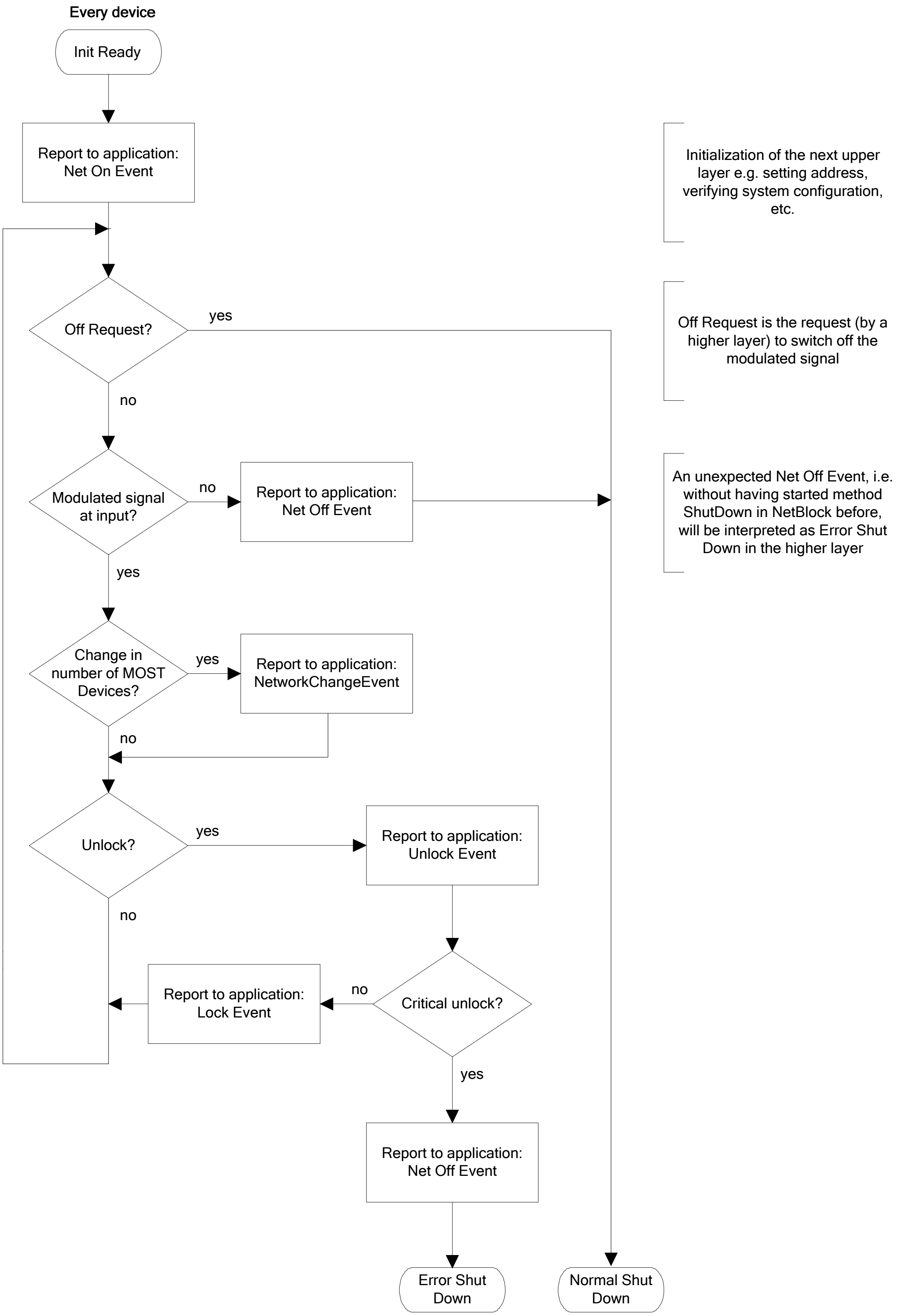

Every device
Init Ready
Report to application: Net On Event
Initialization of the next upper layer e.g. setting address, verifying system configuration, etc.
Off Request?
yes
no
Off Request is the request (by a higher layer) to switch off the modulated signal
Modulated signal at input?
no
Report to application: Net Off Event
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
yes
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 3-9: 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.

The ring break diagnosis process can be started by various triggers, which must be chosen and implemented by the System Integrator. One possible way is to start the ring break diagnosis by disconnecting the System from the power source for a short time. In this case, ring break diagnosis is entered when signal SwitchToPower of the SwitchToPowerDetector indicates that the device was connected to power first time (e.g., after reconnection of the car's battery). If SwitchToPower is used to trigger ring break diagnosis, 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 TimingMaster at the end of ring break diagnosis (since there was no modulated signal at its input).

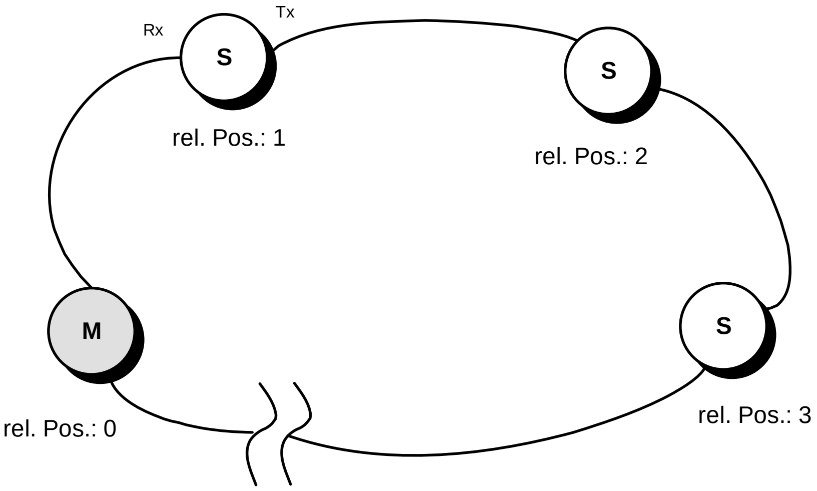


Figure 3-10: Localizing a fatal error with the help of ring break diagnosis.

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

Table 3-11: Events in state NetInterfaceRingBreakDiagnosis

Table 3-12 lists the possible results of ring break diagnosis and the corresponding events.

Ring Break Diagnosis Result	**Description**	**Event**
DIAG_OK	No error detected. Transition to NetInterfaceNormalOperation.	Diagnosis Ready
DIAG_MULTIMASTER	No fatal error detected. But the NetInterface, which was originally configured in TimingMaster mode, has been re-configured to Slave mode, because there is another TimingMaster in the network. Transition to NetInterfaceNormalOperation. At next initialization of NetInterface, the NetInterface has to be configured as timing Slave, to prevent event "Init Error Shut Down".	Diagnosis Ready
DIAG_ALLSLAVE	Fatal error: No TimingMaster existing.	Diagnosis Error Shut Down
DIAG_POS	Ring break detected. The result indicates the relative position to the ring break.	Diagnosis Error Shut Down
DIAG_POOR	Fatal error, but there is no valid relative position available, since stable lock could not be established	Diagnosis Error Shut Down

Table 3-12: Ring break diagnosis results

During ring break diagnosis a device stays configured as TimingMaster until it recognizes a modulated signal 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 TimingMaster 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 TimingMaster under normal operation conditions, recognizes modulated signal at its input, it is configured as Slave (bypass enabled). The 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.

MOST25
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.

MOST50
A stable lock is signaled by the existence of the System Lock flag in the administrative area of the MOST frame. In this case, 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 Init Ready event. After that, all high level initializations must be performed (building of the Central Registry, address initialization, notification...).

If the ring could not be closed because of a ring break, devices should not be restarted by incoming modulated signal until $t_{Diag\_Restart}$ has passed.

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

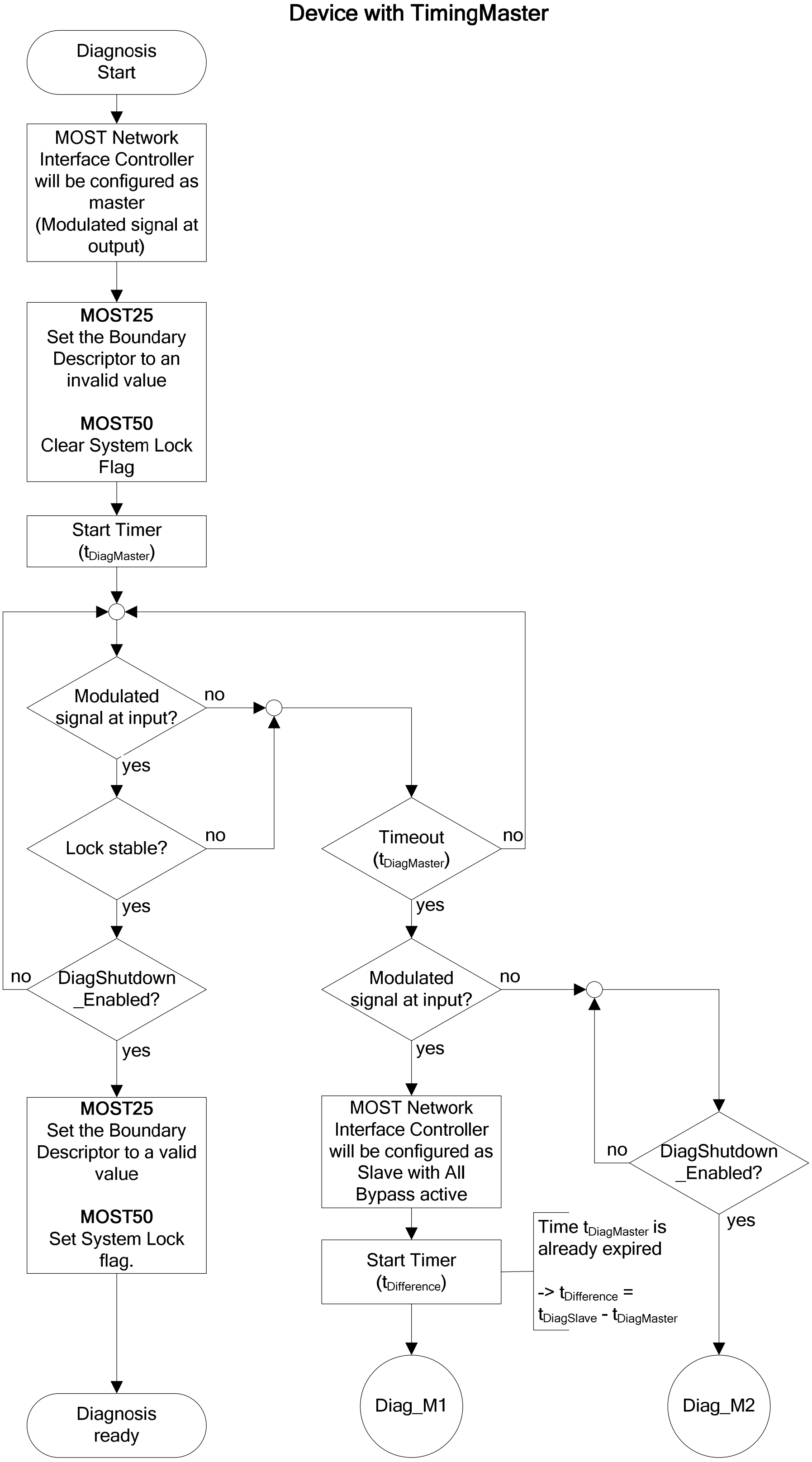

Device with TimingMaster
Diagnosis Start
MOST Network Interface Controller will be configured as master (Modulated signal at output)
MOST25
Set the Boundary Descriptor to an invalid value
MOST50
Clear System Lock Flag
Start Timer ($t_{DiagMaster}$)
Modulated signal at input?
no
yes
Lock stable?
no
yes
DiagShutdown _Enabled?
no
yes
Timeout ($t_{DiagMaster}$)
no
yes
Modulated signal at input?
no
yes
MOST25
Set the Boundary Descriptor to a valid value
MOST50
Set System Lock flag.
MOST Network Interface Controller will be configured as Slave with All Bypass active
Start Timer ($t_{Difference}$)
Time $t_{DiagMaster}$ is already expired
-> $t_{Difference}$ = $t_{DiagSlave}$ - $t_{DiagMaster}$
DiagShutdown _Enabled?
no
yes
Diagnosis ready
Diag_M1
Diag_M2

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

Device with Timing Slave

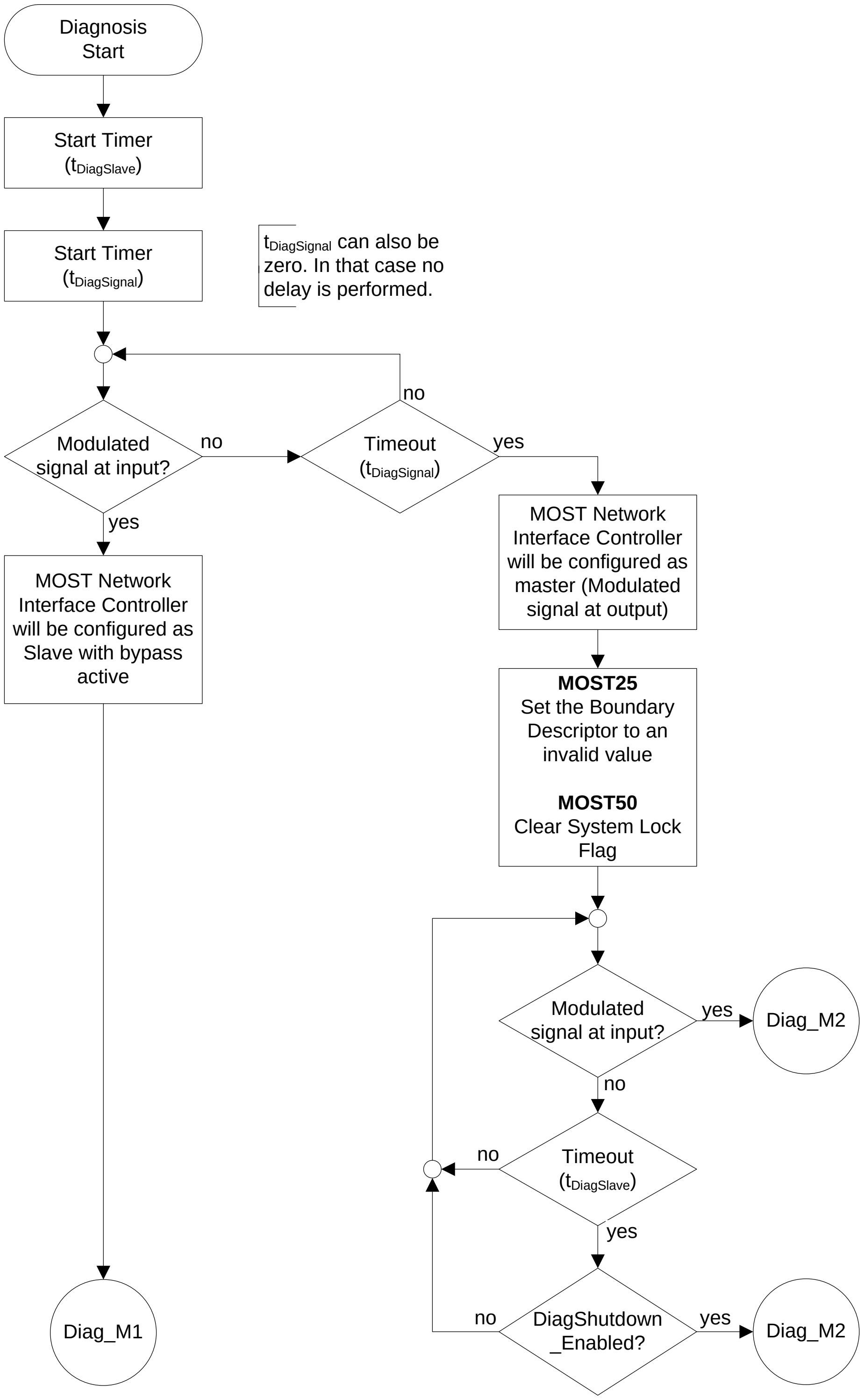

Diagnosis Start
Start Timer ($t_{DiagSlave}$)
Start Timer ($t_{DiagSignal}$)
$t_{DiagSignal}$ can also be zero. In that case no delay is performed.
Modulated signal at input?
no
yes
Timeout ($t_{DiagSignal}$)
no
yes
MOST Network Interface Controller will be configured as Slave with bypass active
MOST Network Interface Controller will be configured as master (Modulated signal at output)
MOST25 Set the Boundary Descriptor to an invalid value
MOST50 Clear System Lock Flag
Modulated signal at input?
yes
Diag_M2
no
Timeout ($t_{DiagSlave}$)
no
yes
DiagShutdown _Enabled?
no
yes
Diag_M2
Diag_M1

Figure 3-12: Behavior during ring break diagnosis in a Slave (part 1)

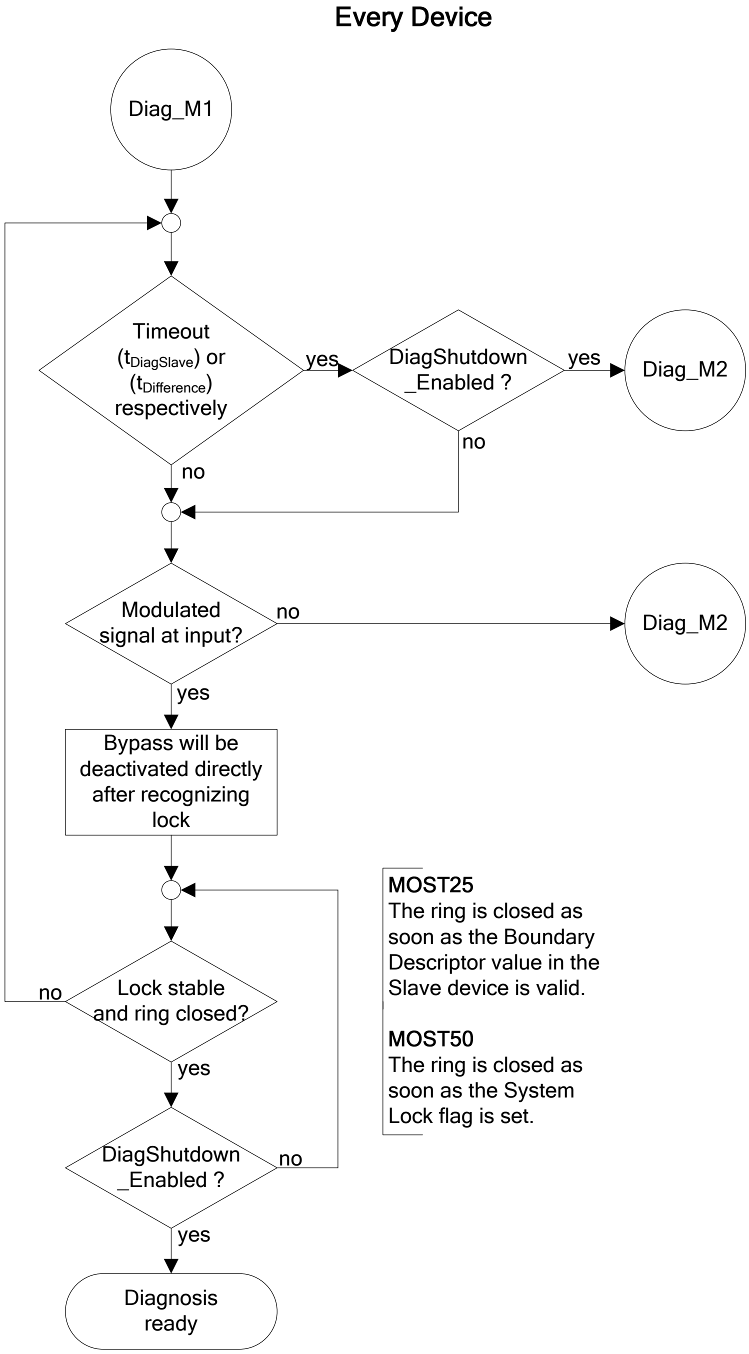


Figure 3-13: Behavior during ring break diagnosis in a TimingMaster and Slave (part 2)

Every Device

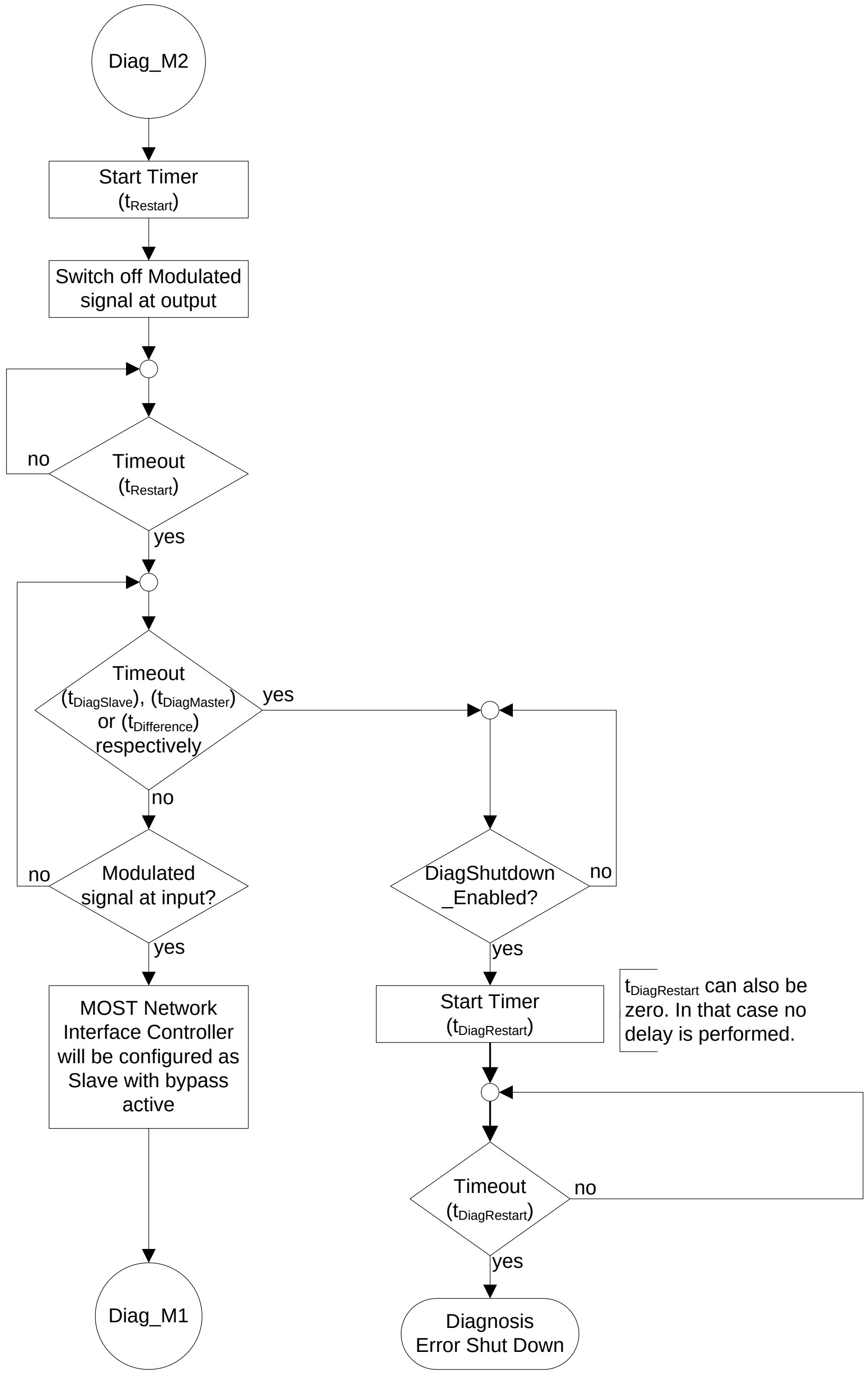

Diag_M2
Start Timer
($t_{Restart}$)
Switch off Modulated signal at output
no
Timeout
($t_{Restart}$)
yes
Timeout
($t_{DiagSlave}$), ($t_{DiagMaster}$)
or ($t_{Difference}$)
respectively
yes
no
no
Modulated signal at input?
yes
DiagShutdown
_Enabled?
no
yes
MOST Network Interface Controller will be configured as Slave with bypass active
Start Timer
($t_{DiagRestart}$)
$t_{DiagRestart}$ can also be zero. In that case no delay is performed.
Timeout
($t_{DiagRestart}$)
no
yes
Diag_M1
Diagnosis
Error Shut Down

Figure 3-14: Behavior during ring break diagnosis in a TimingMaster and Slave (part 3)

3.2.3 Secondary Node

MOST50
This entire section does not apply to MOST50. Secondary Nodes do not exist in MOST50.

For some applications it can be useful to integrate two MOST Network Interface Controllers into one device, a Primary Node and a Secondary Node. A Secondary Node does not contain any FBlocks. In case of receiving any request, it returns an error (Error Secondary Node. Please refer to section 2.3.2.5.1 on page 36). 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 FBlocks 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 explicitly does 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 FBlock 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.

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 micro controller. That node, where Control Messaging is handled (and where the Network Service is running mainly) is called “Primary Node”. There are three scenarios:

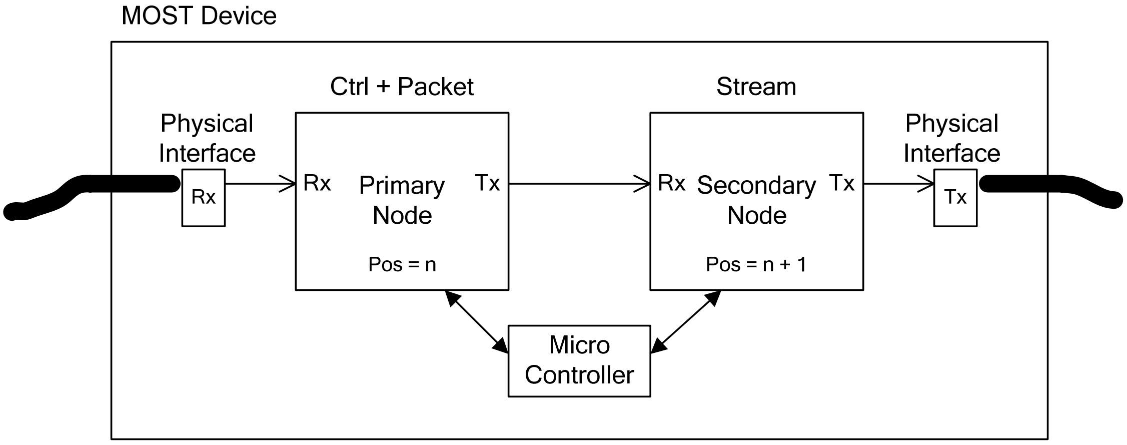
- Scenario 1:
  Primary Node : Ctrl + Packet
  Secondary Node : Stream

- 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. Also, the address of the Secondary Node should be determined and initialized.

3.2.3.1 Scenario 1



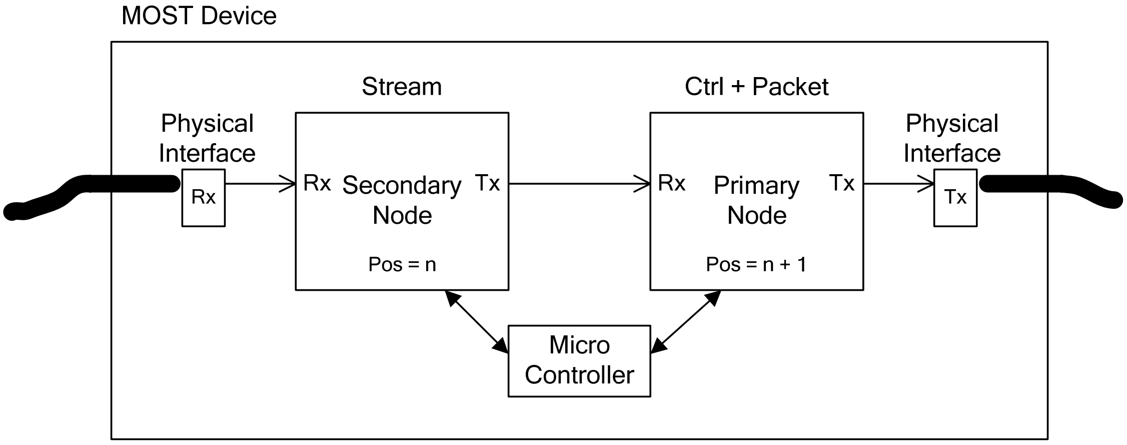
Primary Node	
- SP Parallel Async.	CP Serial
- SP Parallel Async.	CP Parallel

Secondary Node	
- SP Serial,	CP Serial
- SP Parallel Sync.	CP Serial
- SP Parallel Sync.	CP Parallel

Figure 3-15: 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 3-15 shows, which configurations for Source Data Port (SP) and Control Port (CP) are available.

3.2.3.2 Scenario 2



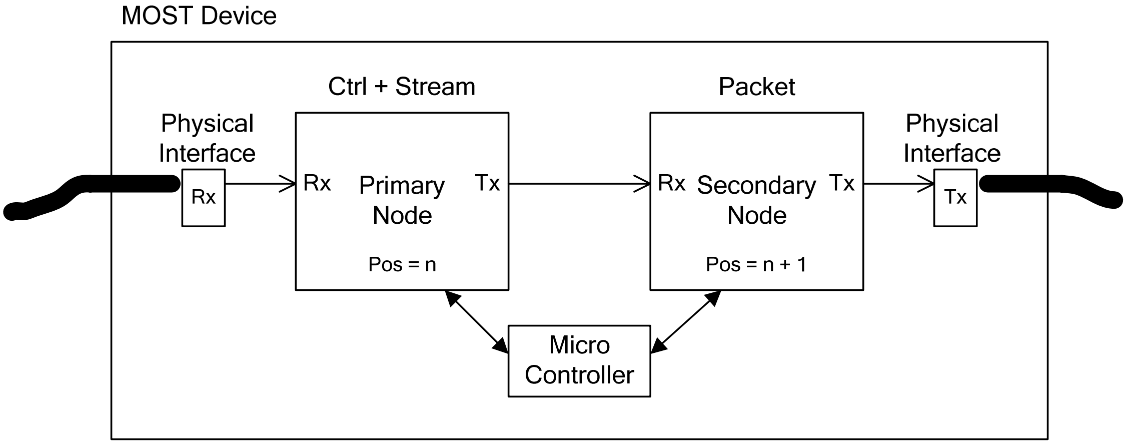
Secondary Node
- SP Serial, CP Serial
- SP Parallel Sync. CP Serial
- SP Parallel Sync. CP Parallel

Primary Node
- SP Parallel Async. CP Serial
- SP Parallel Async. CP Parallel

Figure 3-16: 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 TimingMaster. Figure 3-16 shows which configurations for Source Data Port (SP) and Control Port (CP) are available.

3.2.3.3 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 3-17: 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 3-17 shows which configurations for Source Data Port (SP) and Control Port (CP) are available.

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 switching on a modulated signal. In principle the network can be awakened by any node. The permission 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 PermissionToWake, 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).

By default, the property ”PermissionToWake“ of the NetBlock controls the permission to wake the network via the MOST signal (opt./electr.). Optionally, the System Integrator can define, that PermissionToWake additionally controls the permission to wake via a separate electrical wake-up line.

The value of NetBlock.PermissionToWake is never changed automatically. It can only be changed by using

```
NetBlock.PermissionToWake.SetGet(On/Off)
```

The property NetBlock.PermissionToWake of the device containing the system's PowerMaster should not be set to OFF.

The PermissionToWake property must be stored as long as the device is connected to power. It is up to the System Integrator to decide whether it must be stored when the device is removed from power.

”CapabilityToWake“ indicates the ability of the device to wake the network via the MOST signal (opt./electr.). It does not imply the ability in respect to an additional electrical wake-up line. “CapabilityToWake”, which is mandatory, is reported through NetBlock.DeviceInfo.Status with ID 0x07.

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 a separate 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 a modulated signal at the output of the device. Every node that recognizes the modulated signal at its input switches on a modulated signal at its output and initializes. In this way the modulated signal travels from node to node until the entire network is awake.

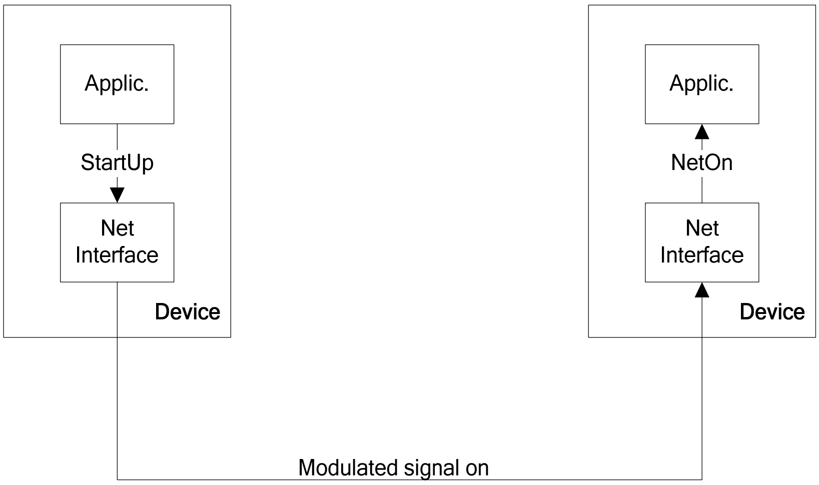


Figure 3-18: Example (2 devices) for waking of the MOST network via a modulated signal on the network

3.2.4.2 Network Shutdown

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

All devices, except the one containing the PowerMaster, switch off the modulated signal 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 modulated signal to be switched off.

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

If an FBlock desires to 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 “modulated signal off” travels around the ring in the same way as “modulated signal on” when waking the network. After a certain delay time $t_{PwrSwitchOffDelay}$ the nodes change to sleep mode.

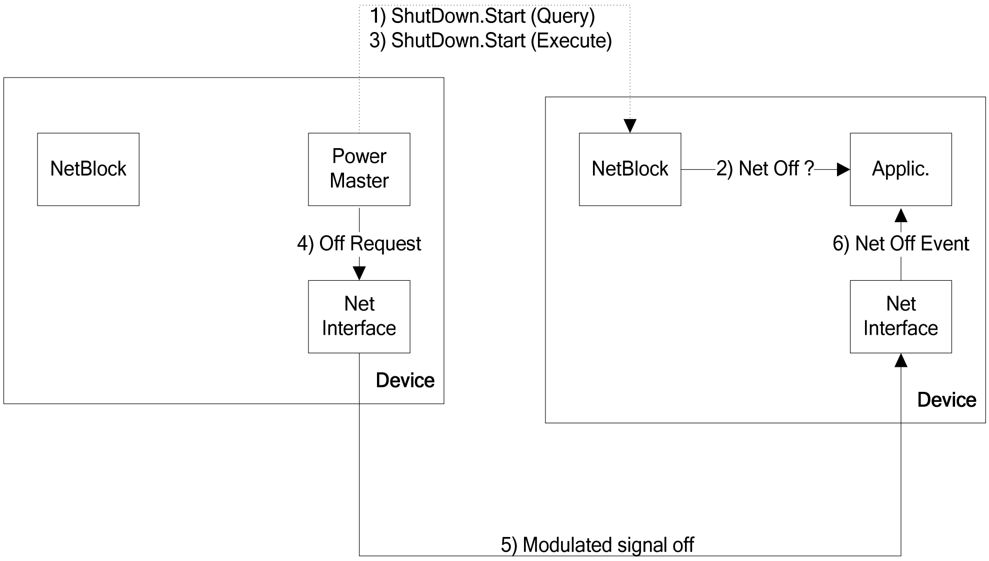


Figure 3-19: Switching off MOST Network via starting method ShutDown in every NetBlock, and signaling to every application, and switching off modulated signal

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

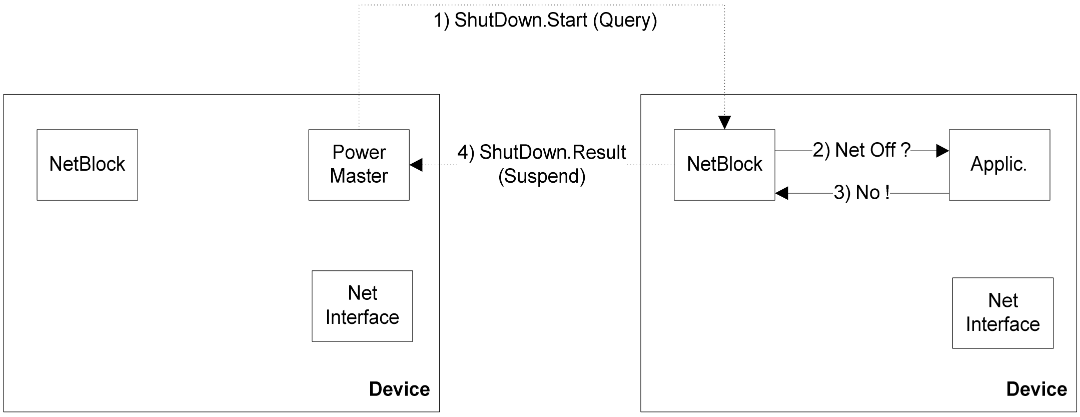


Figure 3-20: Prevention of switching off MOST Network via ShutDown.Result (Suspend)

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

3.2.4.3 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 NetworkMaster Shadow has to keep track of the current System State even while the device is in Device Shutdown state.

The NetBlock.Shutdown.Start message is used to bring a device into or out of Device Shutdown. When managing Device Shutdown, this message may be sent to one device or a group of devices, as opposed to Network Shutdown where this message has to be broadcast. The behavior of a device during Network Shutdown is not affected by whether the device is in Device Shutdown or not. Refer to section 3.2.4.2 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 FBlocks are communicating with FBlocks on other devices. It is also useful if the PowerMaster wants to shut down a group of devices but only if the whole group is ready.

1. The PowerMaster sends Shutdown.Start(Query) to a single device or a group of devices.
2. The PowerMaster will wait for $t_{Suspend}$ 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 streaming outputs.
3. The device unregisters itself from notification matrices in other devices, if any.

The device unregisters its FBlocks by sending an FBlockIDs.Status with an empty FBlockIDList.
The NetworkMaster broadcasts the device’s invalid FBlocks.
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 FBlocks using FBlockIDs.Status(FBlockIDList).
4. When NetworkMaster reports the new FBlocks 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 FBlocks using FBlockIDs.Status(FBlockIDList).
3. When NetworkMaster reports the new FBlocks 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 a high level:

- **Failure of a Network Slave Device:** Error that leads to either a reset of the whole device or reset of the failing application.
- **Critical Voltage:** The NetInterface works normally, the device can communicate. On a recovery from this state, the network does not need to be initialized again.
- **Over-Temperature:** Depending on the temperature, four different alert levels are defined.

Also, errors may occur on a lower level:

- **Fatal Error:** Error that leads to the interruption of the ring, to the breakdown of the network, or that means the network cannot be initialized (super- or sub-voltage, ring break, defect Physical Interface unit).
- **Unlock:** The PLL of the MOST Network Interface Controller is no longer locked. A ring break is not necessarily the inevitable conclusion of this error.
- **Network Change Event:** One of the nodes in the network has activated or deactivated its bypass, which means it “disappears” or “appears” as a new node.
- **Low Voltage:** The voltage of one or more devices is too low to maintain operation of the NetInterface.

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 Streaming Signals:** In opposite to the packet data area and the Control Channel, there is no data securing for the streaming area. 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 streaming 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 streaming connections on the Network are not removed, except in case of a fatal error or a Network Change Event, which leads to the NetworkMaster sending out Configuration.Status(NotOK).

3.2.5.1 Fatal Error

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

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

3.2.5.1.1 Handling of Modulated Signal Off

If a device recognizes at its input that the modulated signal 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 the modulated signal 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 the modulated signal again after tRestart has expired, if the vehicle’s status requires it. It wakes the network in the normal way and by that a re-initialization is done.

If the modulated signal 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 modulated signal reappears within $t_{PwrSwitchOffDelay}$, the system is re-initialized like when waking up after sleep mode. The only difference is that within the devices power supply, micro controller and operating system need not be re-initialized.

3.2.5.1.2 Waking

If a fatal error occurs while an application tries to wake the network, “modulated signal 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.

3.2.5.1.3 Operation

If there is a fatal error during normal operation, “modulated signal off” propagates through the entire ring. This is handled as described above. In case the power status of the vehicle requires it, the PowerMaster tries to wake the network after $t_{Restart}$. So the handling of a fatal error during waking needs to be performed (see above).

3.2.5.2 Unlock

An unlock occurs when a Slave device cannot lock onto the modulated signal of the MOST Network.

Causes for this may be that two TimingMasters in one ring are working against each other or if a TimingMaster does not receive a comprehensible signal.

Another cause can be that the modulated signal at a node's input is too weak or a node opens or closes its bypass. Every node downstream from the location that caused the unlock, up to the TimingMaster, recognizes the unlock. The nodes downstream of the TimingMaster up to the location that caused the unlock do not recognize the unlock. On an unlock, data errors occur. Based on its securing mechanism, the Control Channel is relatively insensitive to short unlocks.

If an unlock would occur all applications must secure its in- and output signals (e.g. an amplifier mutes the outputs).

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 causes 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 3.8 on page 197).

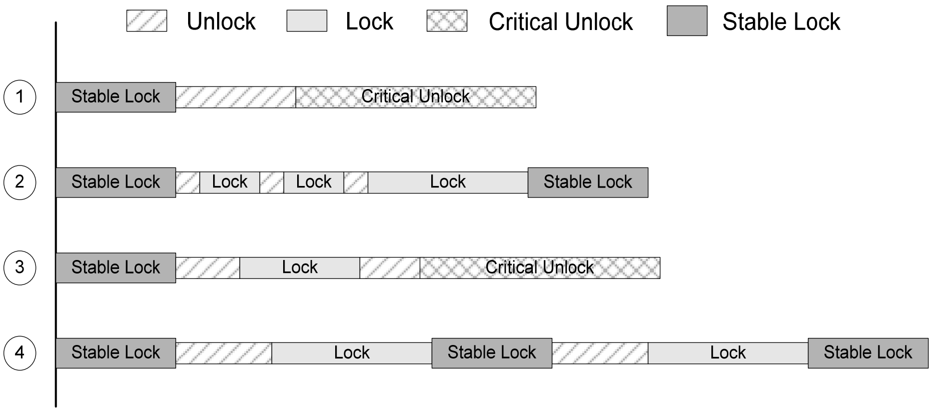


Figure 3-21: Examples of the behavior when unlocks occur.

1. The first example shows an unlock that persists longer than t_{Unlock}. This result in a critical unlock (Error Shutdown event).

2. A series of short unlocks with an accumulated time less than t_{Unlock} will not lead to a critical unlock.

3. Two unlocks with an accumulated time that exceeds t_{Unlock}. This leads to a critical unlock.

4. The unlocks are almost as long as t_{Unlock}. The example shows that the system can withstand a series of long unlocks, provided that a lock time of at least t_{Lock} is interspersed.

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

3.2.5.3 Network Change Event

A Network Change Event (NCE) is defined as a detected change of the Maximum Position Information transmitted cyclically on the Network.

If a device opens or closes its bypass, that is, enters or leaves the Network, the Maximum Position Information changes (except for the case when one device enters and another device leaves the network in a very short time interval[1]).

Disturbances of the Maximum Position Information can occur, for example, because of an unlock.

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 streaming 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

3.2.5.4 Failure of a Network Slave Device

The failure of a Network Slave device can be divided into two scenarios:

1. **A Network Interface Controller failure**, which leads to a reset of the whole device; that is, Network Interface Controller, EHC etc. (bottom-up reset).
2. **An Application failure**. The Network Interface Controller is still OK, hence only the affected application(s) needs to be restarted.

3.2.5.4.1 Failure of the Network Interface Controller

If a device experiences an internal failure of the Network Interface Controller the whole device must be re-initialized.

3.2.5.4.2 Failure of an Application

Every application must be able to handle the case where one of its communication partners does not respond and safely terminate the parts of the program that depend on this communication.

By implementing a watchdog in each device, a long “hanging” of an application should be avoided.

It may happen that single processes in a device are hanging (but not the entire device, as in section 3.2.5.4.1, 'Failure of the Network Interface Controller’), and that those processes need to be restarted. In case this failure stops an entire, or even several FBlocks, the device has to un-register those FBlocks in the Central Registry 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 states, that only those FBlocks contained in FBlockIDList are available. The NetworkMaster updates the Central Registry and broadcasts immediately after the reception of such an un-registration:

```
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 is the list of those FBlocks that are invalid. Therefore, all applications have the required information and can terminate functions depending on the invalid FBlocks.
Note that the device must not communicate until it receives a Configuration.Status message.

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

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

The NetworkMaster registers these FBlocks in the Central Registry and broadcasts immediately after having received the registration:

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

Here, DeltaFBlockIDList is the list of the new FBlocks.

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

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

3.2.5.5 Supply Voltage

The definition of voltage levels and more information can be found in section 4.6 on page 209. 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 streaming 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 modulated signal 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 “modulated signal on” at its input, or by the demand for communication from its own application. It changes to mode DeviceNormalOperation via the standard initialization process. The low voltage reset leads a device to normal behavior.

By opening 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.

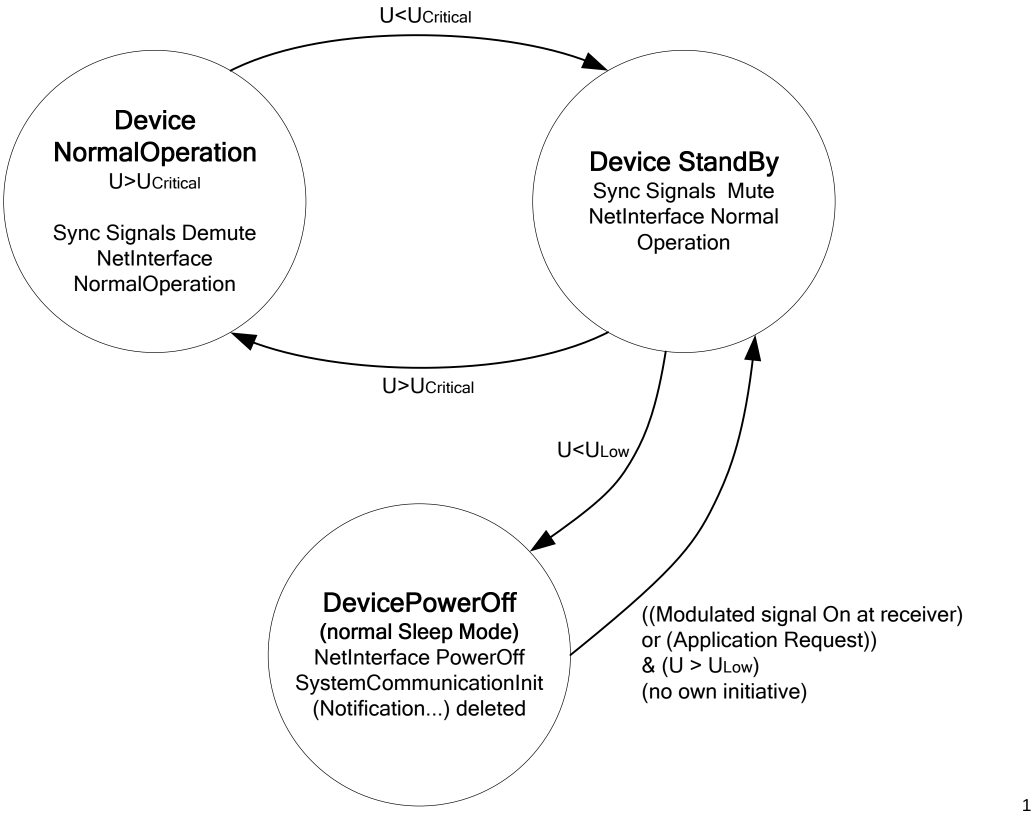


Figure 3-22: Behavior of a device depending on supply voltage

---

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

3.2.5.6 Over-Temperature Management

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 section applies 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.

3.2.5.6.1 Levels of Temperature Alert

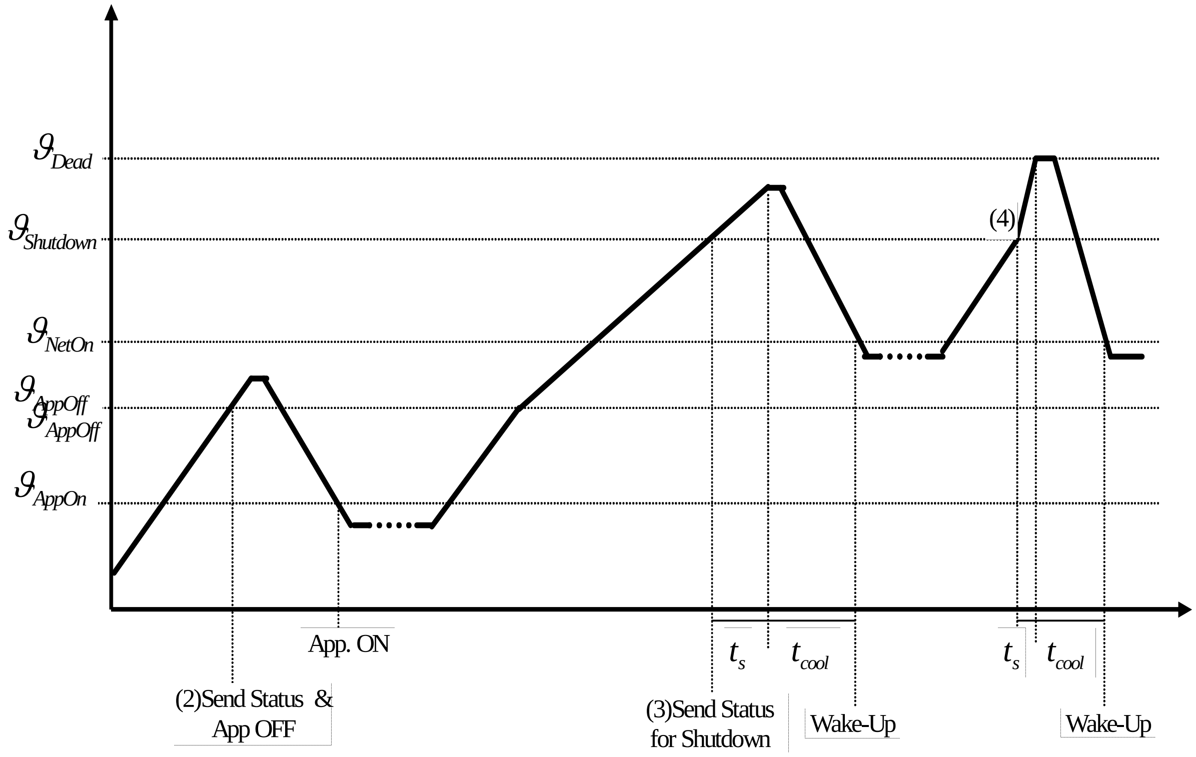


Figure 3-23: Alert Levels

Figure 3-23 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 Physical Interface unit or other components needed for MOST functionality), the device could decide to shut down this specific application. The FBlock 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 PermissionToWake 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 modulated signal 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.

3.2.5.6.2 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, that is, 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. As 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.

One of the methods b) or c) must be supported by the PowerMaster.

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 PermissionToWake 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 lasts no longer than approx. 150ms, so if the minimum interval between such attempts ($t_{WaitAfterOvertempShutDown}$) is, for example, one minute, the short phase of operation 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 NetworkMaster ensures secure communication between applications over the MOST Network. This section describes the conditions that must be met by the NetworkMaster 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 Init Ready event.

3.3.1.1.1 Initialization of the Network

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

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

Setting the System State to NotOK resets the system from a network point of view, that is, any network related information is reset in all Network Slaves. This event prevents the same collisions to occur more than once.

- If there is no valid logical node address available at System Startup the NetworkMaster 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 the NetworkMaster starts to scan the system.

The NetworkMaster will set the System State to NotOK whenever an error is caused by a Network Slave registration. The NetworkMaster will perform a System Scan as described in section 3.3.3.4.

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

3.3.1.1.2 Initialization on Application Level

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

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

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

3.3.1.2 General Operation

3.3.1.2.1 Finding Communication Partners

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

A Controller device may store the information concerning its communication partners in a Decentral Registry. The benefit of having a Decentral Registry is that the 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 NetworkMaster sets the System State to NotOK.

3.3.1.2.2 Network Monitoring

The NetworkMaster monitors the system for changes and errors. When a Network Change Event is detected, the NetworkMaster must find out if a device has entered or left the network. It scans the network and reports any new information to all 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 NetworkMaster 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 FBlocks at any time; these changes have to be reported to the NetworkMaster. The NetworkMaster then updates the Central Registry and informs all Network Slaves.

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

The NetworkMaster builds the Central Registry by collecting logical node addresses and FBlock 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 NetworkMaster distributes the System State of the network to the Network Slaves by broadcasting Configuration.Status messages. The state diagram in Figure 3-24 shows the System States and which events affect the states.

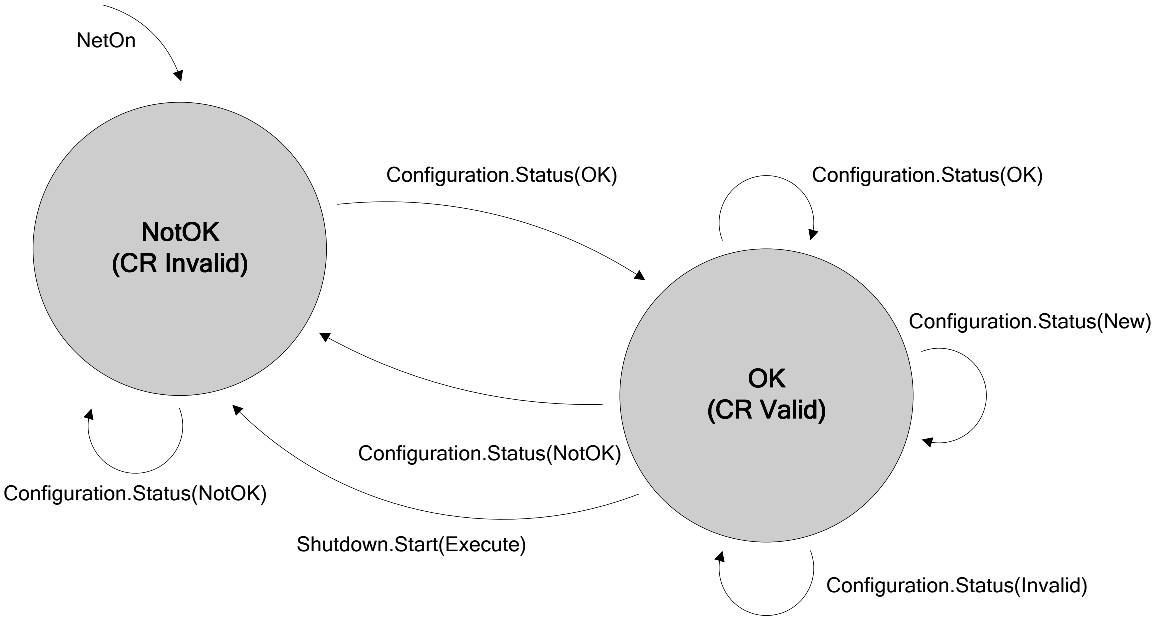

NetOn
NotOK
(CR Invalid)
Configuration.Status(OK)
Configuration.Status(OK)
Configuration.Status(New)
OK
(CR Valid)
Configuration.Status(NotOK)
Configuration.Status(NotOK)
Shutdown.Start(Execute)
Configuration.Status(Invalid)

Figure 3-24: States of the network are shown, as well as the status of the Central Registry[1]

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

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

---

[1] Shutdown.Start(Execute) will not cause the re-calculation of logical node addresses (see 3.3.2.1).

3.3.2.1 System State NotOK

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

Also, the system is regarded as being in System State NotOK after NetBlock.Shutdown.Start(Execute) has been broadcast. An optional delay may be specified on a per-system basis between the broadcast of this message and the point in time where the change of state becomes effective. 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 NetworkMaster - 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 3-13: Events in System State NotOK (refer to Figure 3-24)

3.3.2.2 System State OK

While in System State OK, the Central Registry is valid. So the exact set of FBlocks in the system, each with its attributes InstID and DeviceID, is defined. Therefore, application communication (i.e. messages with FBlockIDs other than NetBlock or NetworkMaster are allowed) may take place on Control and Packet Data 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 3-14: Events in System State OK (refer to Figure 3-24)

Table 3-14 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 Matrix.
- Destroy all windows on LongArrays.
- Every FBlock containing streaming sinks: set the Mute property to "ON" for all sinks and disconnect them.
- Every FBlock containing streaming sources: All sources must route zeroes (signal mute) to its channels for a time $t_{CleanChannels}$. After time $t_{CleanChannels}$ all streaming 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 NetworkMaster

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

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

3.3.3.1 Setting the System State

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

The NetworkMaster must do the following when setting the System State to OK:

1. Broadcast Configuration.Status(OK).
2. Trigger initialization of applications in own device.
3. Continue to maintain the Central Registry.

3.3.3.1.2 Setting the System State to NotOK (Network Reset)

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

The NetworkMaster 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 NetworkMaster 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 FBlocks of each device:

RxTxLog	RxTxPos	FBlockID	InstID
0x0100	0	AudioDiskPlayer	1
		NetworkMaster	10
		ConnectionMaster	1
0x0101	1	AudioDiskPlayer	2
0x0102	2	AM/FMTuner	1
		AudioTapeRecorder	1
0x0103	3	AudioAmplifier	2
Etc.			
MaxNode	MaxNode	HumanMachineInterface	1

Table 3-15: Example of a Central Registry

3.3.3.2.1 Purpose

The Central Registry is used when the NetworkMaster checks the system configuration and when devices are searching for communication partners.

3.3.3.2.2 Contents

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

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

3.3.3.2.4 Responding to Requests for Information from the Central Registry

The NetworkMaster 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.

3.3.3.3 Specific Behavior During System Startup

After the Init Ready event, the NetworkMaster must initialize the system. This process depends on the availability of a valid[1] logical node address.

3.3.3.3.1 Valid Logical Node Address Not Available

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

3.3.3.3.2 Valid Logical Node Address Available

If the NetworkMaster has a valid logical node address at System Startup, it must restore its logical node address and then start a System Scan (section 3.3.3.4).
The NetworkMaster must wait a time $t_{WaitBeforeScan}$ before scanning the system for the first time. This latency time allows the system to stabilize after Init Ready 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 3.1.4.1.

3.3.3.4 Scanning the System (System Scan)

The NetworkMaster scans the system at System Startup and after a Network Change Event (NCE). It may also be instructed to scan the system at any other time.

The NetworkMaster scans the system by requesting the FBlock configuration of each device. The responses from the 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.

The NetworkMaster will set the System State to NotOK whenever an error is caused by a Network Slave registration.

3.3.3.4.1 Configuration Request Description

During a network scan, the NetworkMaster requests NetBlock.FBlockIDs.Get from each Network Slave.

3.3.3.4.2 Addressing

The NetworkMaster 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 NetworkMaster must wait until the expiration of $t_{WaitForAnswer}$ for a reply from a Network Slave. The NetworkMaster 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 NetworkMaster 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 3.8 for more information about timers.

3.3.3.4.5 Network Slave Continuous cause for System State NotOK

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

3.3.3.4.6 Duration of System Scanning

The NetworkMaster 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 NetworkMaster must report the result of the System Scan if it has any new information to distribute, such as a change in System State or changes in the FBlock configuration of one or more Network Slaves. Refer to sections 3.3.3.1 and 3.3.3.6.

3.3.3.5 Invalid Registration Descriptions

The NetworkMaster interprets the incoming registrations and determines if the registration is accepted. The following are considered to be invalid registrations, but all are not considered erroneous since some may be corrected by the NetworkMaster.

3.3.3.5.1 Un-initialized Logical Node Address

If any Network Slave, at any time, registers an un-initialized logical node address (section 3.1.4.1), the NetworkMaster 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 NetworkMaster receives a registration from a Network Slave in which its logical node address is outside of the specified address range, the NetworkMaster 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 NetworkMaster receives a registration from a Network Slave in which its logical node address has already been registered by another Network Slave, the NetworkMaster 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 NetworkMaster is responsible for the uniqueness of functional addresses (combination of FBlockIDs and InstID) within the entire system. The NetworkMaster must try to resolve the issue of two or more Network Slaves registering identical functional addresses.

The NetworkMaster decides a new InstID for the last registered FBlock. It then sets the new InstID in the corresponding Network Slave, by using the logical node address. If the new InstID was accepted by the Network Slave, the NetworkMaster enters the new value into the Central Registry. The NetworkMaster must inform all Network Slaves 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 FBlock is not successful, the NetworkMaster must not include the FBlock into the Central Registry.

3.3.3.5.5 Error Response

A Network Slave that answers a request from the NetworkMaster 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 NetworkMaster 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 NetworkMaster.

This section describes how the NetworkMaster 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 NetworkMaster receives a registration from a Network Slave in which there is one or more FBlocks missing compared to the last registration from the same Network Slave, the NetworkMaster must update the Central Registry and inform all Network Slaves about the missing FBlocks. This is done by broadcasting:

Configuration.Status(Invalid, DeltaFBlockIDList)

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

If the FBlock 0x03 (ConnectionMaster) is unregistered in the Central Registry, a NetworkMaster.Configuration.Status(NotOk) must be broadcast. Afterwards, a System Scan is performed (see section 3.3.3.1.2).

When one or more FBlocks have disappeared the NetworkMaster should inform all Network Slaves about the missing FBlocks 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 NetworkMaster receives a registration from a Network Slave in which there is one or more additional FBlocks compared to the last registration from the same Network Slave, the NetworkMaster must update the Central Registry and inform all Network Slaves about the new FBlock. This is done by broadcasting:

Configuration.Status(New, DeltaFBlockIDList)

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

If this information is gained while scanning the network, the NetworkMaster 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 NetworkMaster 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.

Note:

Configuration.Status(New) with an empty list can only be sent after the scan has been completed.

3.3.3.6.4 Large Updates to the Central Registry in System State OK

If the NetworkMaster 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 NetworkMaster 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 NetworkMaster Requirements

3.3.3.7.1 Network Change Event (NCE)

When an NCE is detected, the NetworkMaster 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 FBlock must be located in a MOST device with a node position address such that it fulfills the requirement of $t_{MPRDelay}$ (refer to section 3.8). The NetworkMaster FBlock is normally located in the same MOST device as the TimingMaster.

3.3.4 Network Slave

All devices that do not contain the NetworkMaster FBlock are called Network Slaves. A Network Slave must keep the NetworkMaster informed about its current FBlock 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 3-16: 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 FBlock entries stored in the Decentral Registry must match the entries of the respective FBlock in the Central Registry. When the NetworkMaster informs of updates to the Central Registry, the Decentral Registry must be updated accordingly for the registered FBlocks.

3.3.4.1.3 Deleting the Decentral Registry

The Decentral Registry must be cleared when the device enters or assumes state NotOk.

3.3.4.2 Specific Startup Behavior

Following the Init Ready event the Network Slave initializes its logical node address, and services requests from the NetworkMaster.

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 3.1.4.1) and services requests from the NetworkMaster until the NetworkMaster 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 NetworkMaster until the NetworkMaster sets the System State.

3.3.4.2.3 Deriving the Logical Node Address of the NetworkMaster

The logical node address of the NetworkMaster 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 3.1.4.1.

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 NetworkMaster

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

The Network Slave must report all FBlocks that are currently active from a network point of view. The FBlocks 0x00 (Network Service), 0x01 (NetBlock) and 0x0F (Enhanced Testability) are neither listed in NetBlock.FBlockIDs.Status messages nor in the Central Registry. In general, it is not recommended to include FBlocks whose InstIDs are coupled with the current node position in the NetBlock.FBlockIDs.Status list or the Central Registry.

If the Network Slave does not have any active FBlocks, it must respond with an empty FBlockIDList.

3.3.4.3.4 Reporting Configuration Changes to the NetworkMaster

When the FBlock configuration of a Network Slave changes, it must report this change to the NetworkMaster; 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 3.2.5.4.

3.3.4.3.6 Failure of a Network Slave Device

This behavior is described in section 3.2.5.4.

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.

Note: The device also has to assume state NotOk when the NetInterface enters normal operation.

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 NetworkMaster. The Configuration.Status (NotOK) implies the system status being NotOK. All other Configuration.Status messages imply the System State is OK, for example,

- Configuration.Status(Ok)
- Configuration.Status(New)
- Configuration.Status(Invalid)
- Configuration.Status(New, <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 NetworkMaster 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 NetworkMaster 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 streaming data associated with any disappearing FBlocks.[1]

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

---

[1] Generally it is more efficient to send individual updates.

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

The NetworkMaster 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 NetworkMaster sets the System State to NotOK, the Network Slave must: Also add to GeneralPlayer?

1. Clear any Decentral Registry.
2. Derive and set the new logical node address (section 3.4.1).
3. Empty Notification Matrix.
4. Destroy all windows on LongArrays.
5. Every FBlock containing streaming sinks: set the Mute property to "ON" for all sinks and disconnect them.
6. Every FBlock containing streaming sources: All sources must route zeroes (signal mute) to its channels for a time $t_{CleanChannels}$. After time $t_{CleanChannels}$ all streaming sources must de-allocate the channels they have allocated and stop routing data to the network.
7. Service requests from the NetworkMaster 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 FBlocks have entered the system and the Central Registry has been updated with the FBlocks supplied in the message. These FBlocks should be added to the Decentral Registry, if they are used by the device.

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

3.3.4.3.15 Reaction to Configuration.Status(Invalid)

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

If a device receives a Config_Invalid with (part of) its own FBlockID(s) and InstID, it has to reinitialize either the announced applications or all applications of the device. If it reinitializes all of its applications, the device sends an empty FBlockID-List to the NetworkMaster.

When reinitializing, the device has to secure the streaming and packet data connections, as well as to clear all notifications.

The device has to clear its routing table and furthermore has to be able to handle the periodic update of the allocation table. After the reinitialization is done, the device has to register its FBlocks again.

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 (*RxTxPos*)**
  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 (*RxTxLog*)**
  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:

RxTxPos = 0x0400 + Pos

Pos = 0 for TimingMaster
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:

RxTxLog = 0x0100 + Pos

The device containing the TimingMaster 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, using the static address range, 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 and can be modified using this function if required. The default procedure for deriving a group address is to take the FBlockID of the FBlock that is most characteristic for the device:

GroupAddress = 0x0300 + FBlockID

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

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

Groups can be built dynamically by modifying group addresses.

If the device is removed from power it is optional to store the group address in non-volatile memory. The group address can be recovered when the system starts up the next time.

If the group address is stored in volatile memory, it is lost if the device loses power for some time. In that case, 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**
NodePosition Address	Get	Controller	NetBlock	Requesting node position address
	Status	NetBlock	Controller	Answer
NodeAddress	Get	Controller	NetBlock	Requesting logical node address
	Status	NetBlock	Controller	Answer
GroupAddress	Get	Controller	NetBlock	Requesting group address
	Status	NetBlock	Controller	Answer
	Set	Controller	NetBlock	Setting group address

Table 3-17: 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. The MOST Network Interface Controller may allow specifying the number of retries and the delay between the retries. Typically, these values should not be changed, however, they can be modified to fine tune the system.

3.4.4 Basics for Automatic Adding of Device Address

Since applications know only functional but not device addresses, a protocol that is transported must be complemented by the device 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:

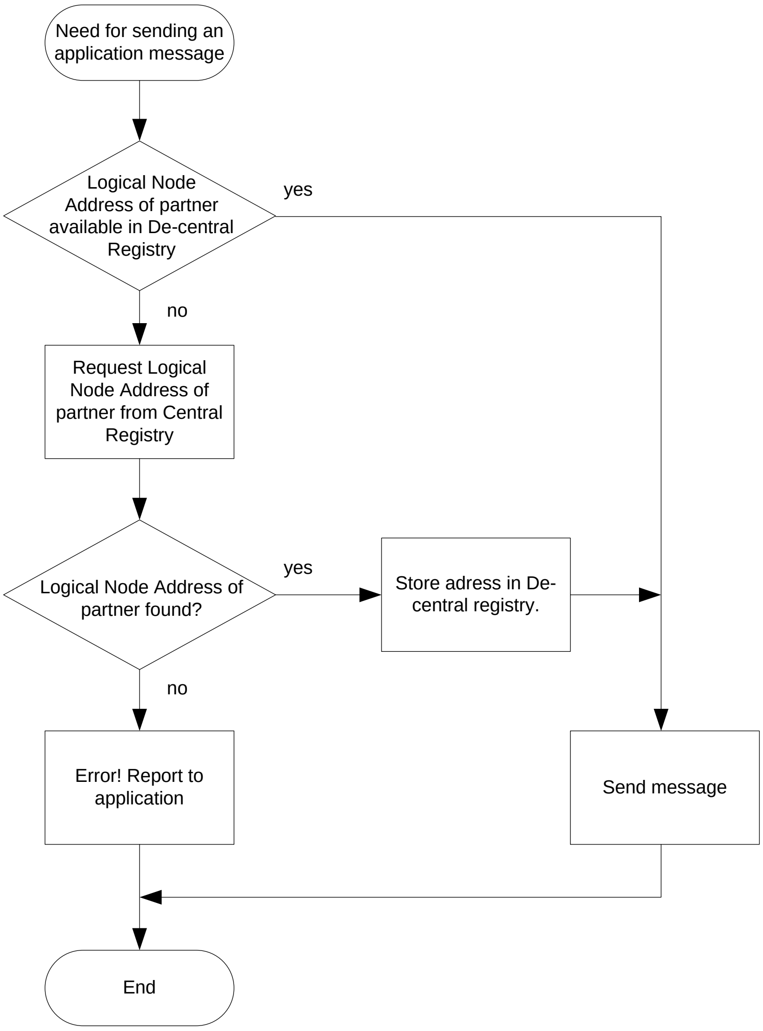

Need for sending an application message
Logical Node Address of partner available in De-central Registry
yes
no
Request Logical Node Address of partner from Central Registry
Logical Node Address of partner found?
yes
Store adress in De-central registry.
no
Error! Report to application
Send message
End

Figure 3-25: Seeking the logical address of a communication partner

3.4.5 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:

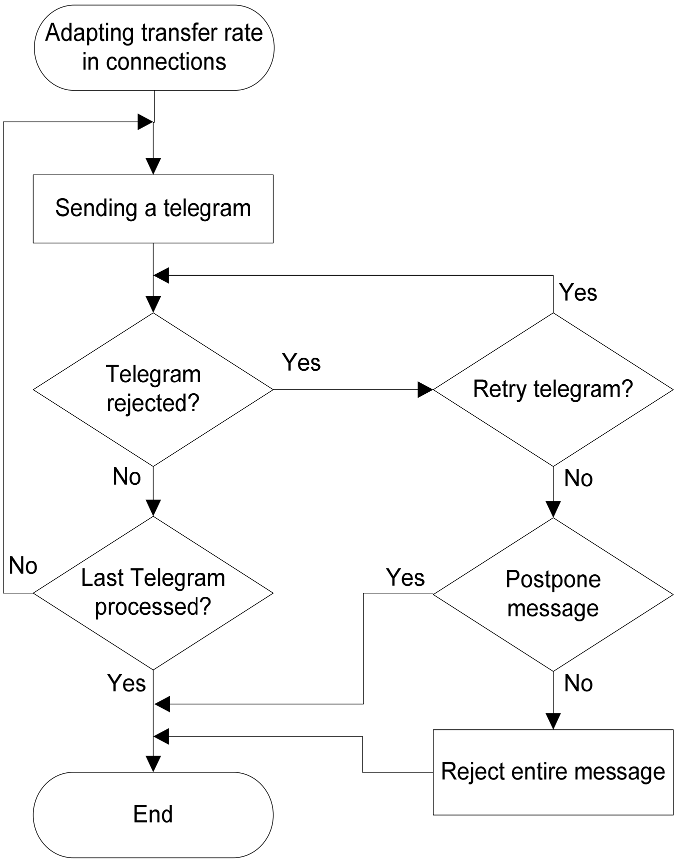


Figure 3-26: Possible mechanism to adapt transfer rates to the speed of a data sink

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, for example, because it is no longer relevant.

3.) The entire message can be postponed, that is, sent later.

3.4.6 MOST Message Services

3.4.6.1 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 (RxTxAdr), and a maximum number of 17 data bytes.

Data area of MOST Network Interface Controller = 17 bytes

16 bits											
RxTxAdr											

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.

3.4.6.2 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:

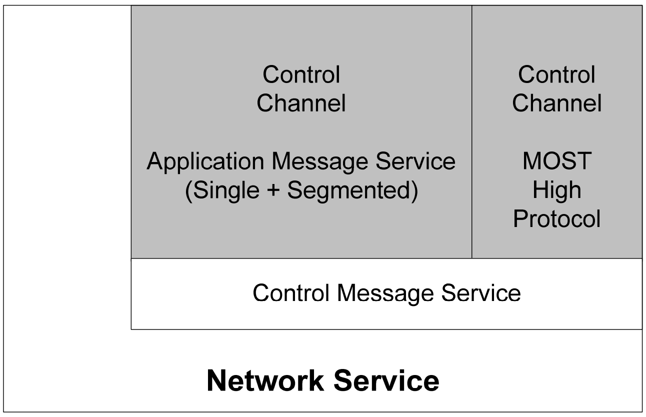


Figure 3-27: Network Service: Services for Control Channel

- **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 28, 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 bytes

16 bits	8 bits	8 bits	12 bits	4 bits	4 bits	4 bits	8 bits	8 bits		8 bits
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.
The telegram length (TelLen) is not an upper bound for the data length instead it is mandatory that the TelLen is the exact length of the valid data.

TelID: Identification of kind of telegram

Meaning	TelID	Data 0 = MsgCnt
Single Transfer	0	Data 0
1st telegram Segmented Transfer	1	0x00
2nd telegram Segmented Transfer	2	0x01
…	2	…
…	2	0xFF
…	2	0x00
…	2	…
(n-1). 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 the number of data bytes that are valid in the telegram.

Data 0-Data 11: Data bytes

3.5 Handling Streaming Data

3.5.1 MOST Network Service API

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

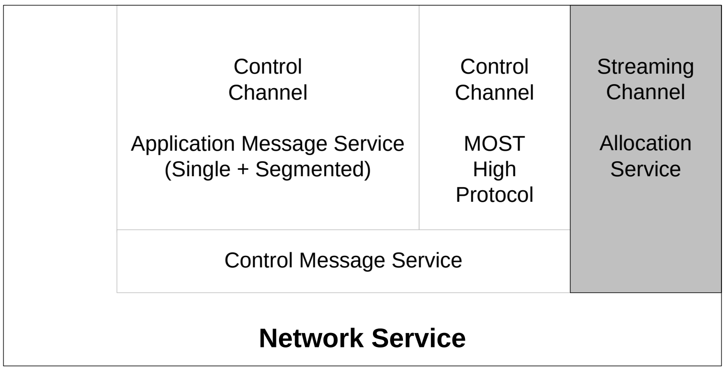


Figure 3-28: Network Service for the Streaming Channel

On the network, streaming and packet data can be transported within a MOST frame. Definition:

- **A *Streaming Channel* corresponds to a byte (8 bits) in the MOST frame**

A certain number of these channels can be used for streaming data transfer.
Several channels can be clustered into a *streaming connection* for an application. Each connection is assigned a *Connection Label*.
The NetInterface connects the channels within a device to the MOST Network.

MOST25
On the network, 60 bytes for streaming and packet data transport are available per frame. A certain number of these channels can be used for streaming data transfer. Here, several channels can be clustered to a streaming 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).

MOST50
Bandwidth is allocated implicitly by creating a socket where the size of the socket denotes the amount. By simply destroying the socket, the allocated bandwidth is de-allocated.
This approach relies on a connection label (16-bit word) combined with bandwidth information (also 16-bit word), instead of a long list of single byte channels.
Furthermore, the routing and the low-level allocation are entirely encapsulated by the Network Interface Controller and of no influence to the application.

3.5.2 Function Block Functions

On the application level, different basic functions in sources and sinks are realized, which serve the administration of streaming connections. They themselves access the routines of the Network Service. The FBlock functions can be categorized into three categories:

- Functions that represent the whole device and are located in NetBlock.
- Functions that provide information about both sources and sinks within an FBlock.
- Functions that deal specifically with only sources or sinks within an FBlock.

3.5.2.1 NetBlock

NetBlock provides the following method to the network:

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

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

Since there can be several FBlocks 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 FBlocks using them.

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

3.5.2.2 General Source / Sink Information

An FBlock containing a source or sink provides the following function:

- **SyncDataInfo**
  Function SyncDataInfo can be used to get information of how many connections the FBlock 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 FBlock:

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

3.5.2.2.1 Streaming Source

Some of the functions that a streaming source provides to the network are shown below. The functions are requested with parameters from the ConnectionMaster and will return a corresponding list of parameters.

- **SourceInfo**
  Property SourceInfo contains detailed information about the kind of streaming source data that the source can handle. The source information is specific for each source number.

  On a request with the SourceNr:

  ```
  Controller -> Slave: FBlockID.InstID.SourceInfo.Get (SourceNr)
  ```

  The following is received:

  ```
  Slave -> Controller: FBlockID.InstID.SourceInfo.Status (SourceNr, DataType,
                                                        [DataDescription])
  ```

  The DataType parameter describes the kind of streaming data stream that is sent by the source. Depending on DataType, a DataDescription may follow. Information about data types can be found in the corresponding Stream Transmission specification.
  Note that DataDescription is basically static, but with one exception: the element ChannelList.

- **SourceName**
  Property SourceName holds the name of the streaming source.
  It is requested with the SourceNr as parameter:

  ```
  Controller -> Slave: FBlockID.InstID.SourceName.Get (SourceNr)
  ```

  The answer is a string containing the name:

  ```
  Slave -> Controller: FBlockID.InstID.SourceName.Status (SourceNr, SourceName)
  ```

- **SourceActivity**[1]
  Some streaming source applications require either to start or to stop transmission of 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 streaming source data stream, the abstract method SourceActivity is defined.

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

  Through parameter Activity = [On/Off/Pause], streaming data transfer can be started, stopped, or paused.

---

[1] The SourceActivity is optional.

After completion of the respective action, the following reply will be generated:

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

There are two approaches of connecting a source to the network, SourceConnect and Allocate.

MOST25
Both approaches are valid.

MOST50
Routing and low-level allocation are entirely encapsulated by the Network Interface Controller and of no influence to the application. The ConnectionMaster cannot apply the SourceConnect approach; the Allocation approach must be used when building streaming connections.
A source drop is detected by the Network Interface Controller on data link layer. This leads to automatic de-allocation of unused bandwidth.
In addition, such an event is signaled to applications that were connected to this channel.

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 ConnectionManager must only use one approach per FBlock. But devices may support both methods.

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 ConnectionManager 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 its channels. 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 TimingMaster is reported.

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

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 Streaming Sink

A FBlock that is used as a sink for streaming data supports similar functions to those for a source. Error handling is also done in an analogous way.

Some of the functions that a streaming sink provides to the network:

- **SinkInfo**
  Property SinkInfo contains detailed information about the kind of streaming sink data that the sink can handle. The sink information is specific for each sink number.
  On a request with the SinkNr:

  ```
  Controller -> Slave: FBlockID.InstID.SinkInfo.Get (SinkNr)
  ```

  The following is received:

  ```
  Slave -> Controller: FBlockID.InstID.SinkInfo.Status (SinkNr, DataType,
                                                         [DataDescription])
  ```

  Information about data types can be found in the corresponding Stream Transmission specification.
  Note that DataDescription is basically static, but with one exception: the element ChannelList.

- **SinkName**
  Property SinkName holds the name of the streaming sink.
  It is requested with the SinkNr as parameter:

  ```
  Controller -> Slave: FBlockID.InstID.SinkName.Get (SinkNr)
  ```

  The answer is a string containing the name:

  ```
  Slave -> Controller: FBlockID.InstID.SinkName.Status (SinkNr, SinkName)
  ```

- **Connect**
  A Controller uses Connect to connect the sink to specified channels.

  ```
  Controller -> Slave: FBlockID.InstID.Connect.StartResult (SinkNr, SrcDelay,
                                                            Channels)
  ```

  SrcDelay is the relative delay to the TimingMaster. 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 streaming data from a sink can stopped by using the property Mute.

  ```
  Controller -> Slave: FBlockID.InstID.Mute.SetGet (SinkNr, Status)
  ```

  Status is On or Off to turn mute on or off.

3.5.2.2.3 Handling of Double Commands

Normally a repeated streaming control command (this means: allocate / deallocate / connect / Disconnect / SourceConnect / SourceDisConnect) should not occur. This handling should be done by the Connection Manager. But in an error case the behavior of the device is defined in the following way:

- **Source methods**
  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.

MOST25
Both the SourceConnect and Allocate approach are valid.

MOST50
Only the Allocate approach is valid.

- **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.

3.5.2.2.4 Order of Streaming Channel Lists

Lists of Streaming Channels must always be in ascending order. Devices that receive a list in any other order must perform error handling rather than trying to re-sort the list.

MOST25
If an unordered Streaming Channel list is encountered, error code 0x06 will be returned.

3.5.2.3 Compensating Network Delay

MOST25
Every node in the ring may generate a deterministic delay for the source data (caused by internal processing). The MOST system 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 TimingMaster. 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 (calculated in number of frames):

$\Delta T_{comp} = \Delta T_{Node} - \Delta T_{Source} - 2$[Samples] for $\Delta T_{Source} < \Delta T_{Node}$
$\Delta T_{comp} = \Delta T_{Network} - \Delta T_{Source} + \Delta T_{Node} - 2$ [Samples] for $\Delta T_{Source} > \Delta T_{Node}$

($\Delta T_{xxx} \equiv$ Contents of the respective register in the chip * 2 Samples Delay)

MOST50
The effective network delay is negligible (delay has to be smaller than 1 µs per node). No network delay compensation is required.

3.6 Handling Packet Data

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 required 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.

MOST25
In every node (MOST Network Interface Controller) the priority for packet data transfer can be assigned. There are different priority levels.

3.6.1 MOST Network Service

The Network Service has a mechanism, called the Packet Data Transmission Service, by which the data packets described above can be sent and received.

3.6.1.1 Securing Data

For certain applications, it may be useful to implement securing mechanisms. For each application it has to be decided whether such mechanisms are required. In that case, a protocol has to be chosen, for example:

- MOST High
- TCP/IP over MAMAC

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 bytes (Data Link Layer 48 Bytes Mode)

16 bits	8 bits	8 bits	12 bits	4 bits	4 bits	12 bits	8 bits	8 bits		8 bits
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 bytes (Alternative Data Link Layer)

16 bits	8 bits	8 bits	12 bits	4 bits	4 bits	12 bits	8 bits	8 bits		8 bits
DeviceID	FBlock ID	Inst. ID	Fkt ID	OP Type	Tel ID	Tel Len	Data 0	Data 1	...	Data 1007

TelID: Identification of telegram type

Meaning	TelID
MOST High Protocol User data	8
MOST High Protocol Control data	9
MAMAC PacketsEthernet frames	A, B

TelLen: = up to 1008 (42)
Specifies the length of the data field, that is, the number of bytes after TelLen

Data X: Data bytes

For securing data, MOST High Protocol is used here.

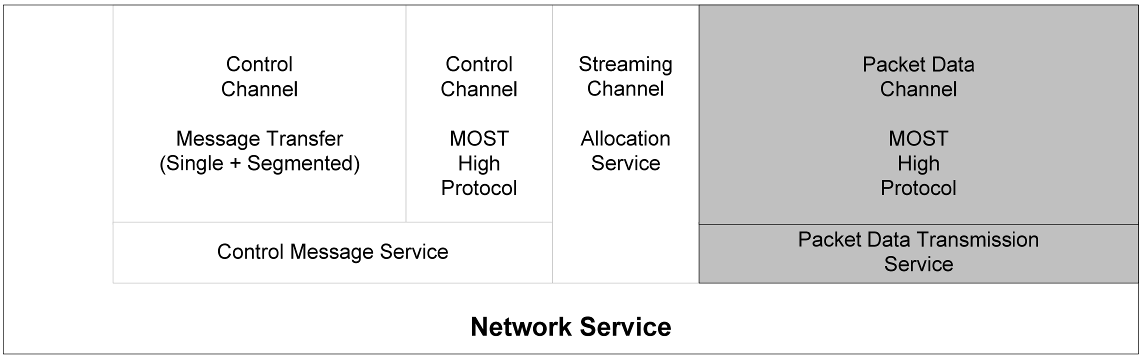


Figure 3-29: Network Service: Services for the Packet Data Channel

3.6.1.2 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 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 Connections

3.7.1 Bandwidth Management

In section 2.1.4 Managing Streaming/Packet Bandwidth, the Boundary Descriptor was introduced.
Since the Boundary is dynamically changeable during runtime, the system can be adopted to the needs which occur in temporary or persistent situations (e.g., data download, transfer of big amounts of data like MP3 or navigation maps). In both situations a change sequence shall be performed without loss of the bus signal.
In order to adapt to the new situation all devices must implement appropriate mechanisms. Therefore they may have to notify itself to the property Boundary of the TimingMaster device.

Adjusting sequence:
For managing the Boundary, the ConnectionMaster provides the method MoveBoundary.

```
Controller -> ConnectionMaster: ConnectionMaster.1.MoveBoundary.StartResult
                                                   (BoundaryDescriptor)
```

With the aid of this method any initiator can request an adjustment of the Boundary. After the ConnectionMaster has received such a request it locks itself and rejects all subsequent connection requests.
Additionally it may inform initiators of rejected connection requests about the reason of the unavailability. After an optional delay it uninstalls all established connections. If all connections are removed, the ConnectionMaster forwards the request for shifting the Boundary to the TimingMaster. This is achieved by setting the property Boundary with OPType SetGet in the device containing the TimingMaster, that is, NetBlock with InstID 0. The TimingMaster internally sets the Boundary to the requested value and re-initializes the low level allocation mechanisms in all devices in the network. If the function Boundary.SetGet returns, the ConnectionMaster forwards the result to the initiator by the respective Result or Error message, unlocks itself and accepts subsequent connection requests. Depending on the configuration the ConnectionMaster may rebuild the previously removed connections.

3.7.2 Streaming Connections

3.7.2.1 ConnectionMaster

Streaming 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 streaming 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 3-18: Functions in ConnectionMaster in conjunction with the administration of streaming connections

Deadlock prevention:

In order to prevent potential deadlocks in the connection building process, $t_{CM\_DeadlockPrev}$ is used. $t_{CM\_DeadlockPrev}$ is started as the Connection Manager makes a request to a source/sink FBlock. If the timer expires the action will be regarded as failed.

This timer should not be used as a maximum time for the source/sink to carry out their respective operations since this must be done much faster; it is merely used to prevent deadlocks and should only be effective in the special cases where a source/sink device has malfunctioned after receiving the command from the Connection Manager.

3.7.2.2 Establishing Streaming Connections

When building a streaming 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.

MOST50
SourceConnect is not available for MOST50.

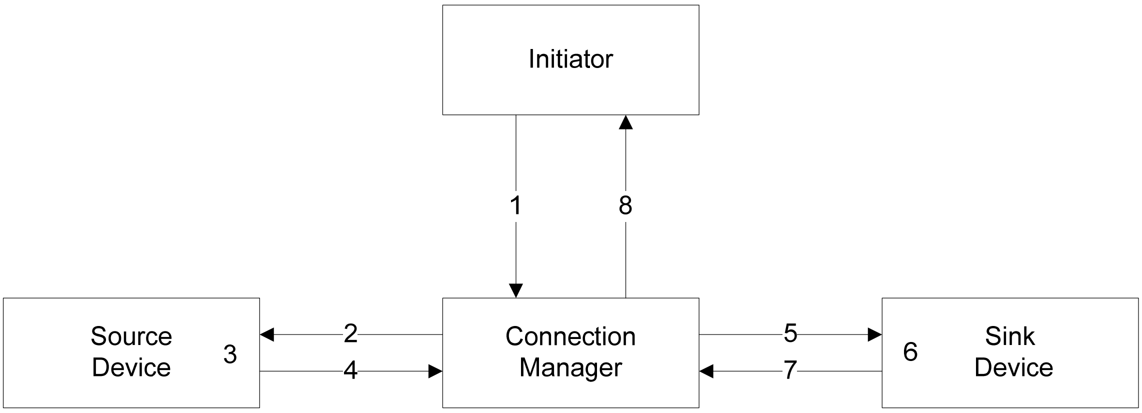


Figure 3-30: Building a streaming connection step by step

Explanation of Figure 3-30**:**

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 streaming 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.
      - TimingMaster is busy on processing other allocation/deallocation requests. The source shall perform 20 retries. If these are not successful, Allocate is regarded as failed by the Connection Manager. The rate at which the TimingMaster 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 are unmade. That means that if an error occurred in the *Connect* process, the source is disconnected and the channels are freed again.

3.7.2.3 Removing Streaming Connections

The initiator terminates a connection previously built by the Connection Manager. The Connection Manager commands the sink to disconnect from the channels that 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.

MOST50
SourceDisConnect is not available for MOST50.

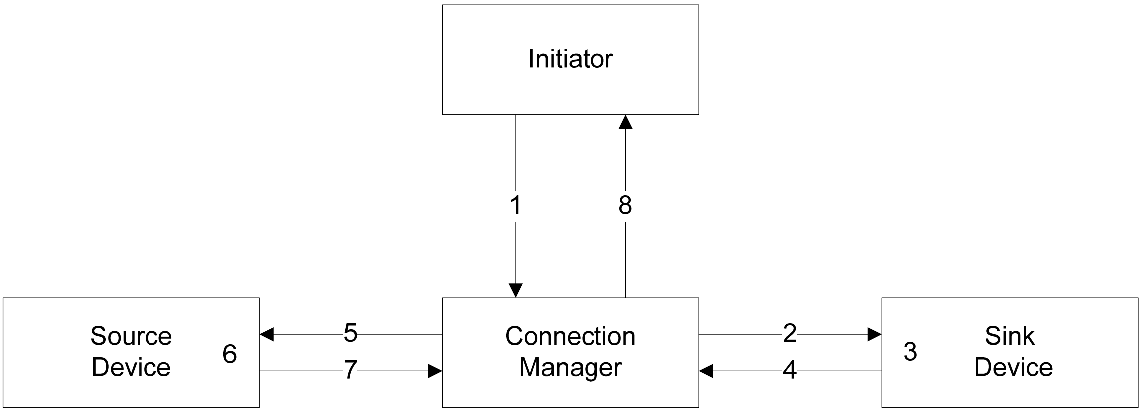


Figure 3-31: Step by step removal of a streaming connection

Explanation of Figure 3-31**:**

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 TimingMaster 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 TimingMaster 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.

3.7.2.4 Supervising Streaming Connections

Every streaming 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 streaming output signals.

3.7.2.4.1 Enabling Streaming 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. If a sink cannot make sure that it receives valid data it must use RemoteGetSource if it cannot make sure that it receives valid data by other mechanisms.

3.7.2.4.2 Source Drops

A source that malfunctions might drop from the network, thus generating a Network Change Event.

If part of the device is still operational, the source device must 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.

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 streaming output. Sinks with automatic noise detection do not need to use this method.

3.8 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 transition to NetOn state.
t_{WakeUp}	–	6	25	ms	C	Maximum time between start of activity at the Rx input of the device and start of activity at the device's Tx output. $(63 \bullet t_{WakeUp}) + t_{WaitNodes} + t_{Lock} + t_{Boundary} < t_{Config}$
$t_{WaitNodes}$	–	–	100	ms	C	Time that may pass between start of activity at the Rx input of a device and the deactivation of its bypass. This timer is valid only when starting up the network.
$t_{Boundary}$	–	–	20	ms	C	**MOST25** Time after which a change of the Boundary must be detected while waiting for the Init Ready event. **MOST50** Not relevant for MOST50.
$t_{WaitBeforeScan}$	100	100	500	ms	T	Time between broadcast of Configuration.Status(NotOk) or an Init Ready event 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 system scans since the network entered the NormalOperation state.
$t_{DelayCfgRequest2}$	10	10	11	s	T	Same as $t_{DelayCfgRequest1}$, but from the 21st attempt on.

Name	**Min Value**	**Typ Value**	**Max Value**	**Unit**	**Type**	**Definition**
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 formula applies: 1. $t_{RestartMin}$ = (Number of nodes) • $t_{ShutDown}$ – 10ms 2. $t_{RestartMax}$ = (Number of nodes) • $t_{ShutDown}$ + 65ms 3. $t_{RestartMin} \le t_{RestartTyp} \le t_{RestartMax}$
$t_{PwrSwitchOffDelay}$	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). If the modulated signal was not switched off within $t_{SlaveShutdown}$, a Slave device may switch off the modulated signal.
$t_{WaitAfterOvertempShutDown}$	20	60	–	s	T	Time the PowerMaster waits after an overtemperature shutdown before it may restart the network.

Name	Min Value	Typ Value	Max Value	Unit	Type	Definition
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 bypass of a device must stay closed after being closed. This timer is not required when starting up the network. It is only required when a node drops out of the network. $t_{Bypass} \leq t_{WaitNodes}$
t_{Answer}	–	–	50	ms	C	Time during which a network Slave must respond to a query by the NetworkMaster.
$t_{WaitForAnswer}$	100	200	700	ms	T	Time a NetworkMaster waits for an answer from a queried Slave.
$t_{ResourceRetry}$	–	–	10	ms	C	Time between attempts of allocating or deallocating streaming 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. The maximum value is a default value and may be adjusted by individual FBlock specifications.
$t_{ProcessingDefault1}$	–	100	150	ms	T	Time a device waits before sending the first Processing message.

---

[1] Devices containing such applications must only be used in node positions where this requirement can be fulfilled.

Name	Min Value	Typ Value	Max Value	Unit	Type	Definition
$t_{ProcessingDefault2}$	100	100	–	ms	T	Time a device waits between sending subsequent Processing messages.
$t_{WaitForProcessing1}$	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 Streaming Channels before it stops using them.
$t_{CM\_DeadlockPrev}$	-	-	1000	ms	T	Timer to prevent deadlocks in the connection building process.
$t_{WaitForNextSegment}$	4975	5000	10015	ms	T	If the next segment of a segmented message does not arrive before this timer elapses, garbage collection is initiated.
$t_{MsgResponse}$	-	-	15	ms	T	Maximum time after which a device must have read a Control Message from the Rx buffer of the MOST Transceiver. This determines the frequency by which the NetServices must be called.
$t_{BoundaryChange}$	100	500	5000	ms	T	Delay between sending the Boundary change notification and start of performing the actual change.

Name	Min Value	Typ Value	Max Value	Unit	Type	Definition
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_Signal}	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 SwitchToPower 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 3-19: Timing Definitions

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, for example, 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:

- Interface area
- MOST function area
- µC area
- Application area
- Power supply area

A MOST network can be awakened by activity on the bus line. They activate a sleep mode if required. If a device is in sleep mode, the power consumption should be reduced as far as possible. A typical value for quiescent current is below 100 µA. Exact parameter and environment conditions, as well as the measurement setup is system-integrator specific. 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.

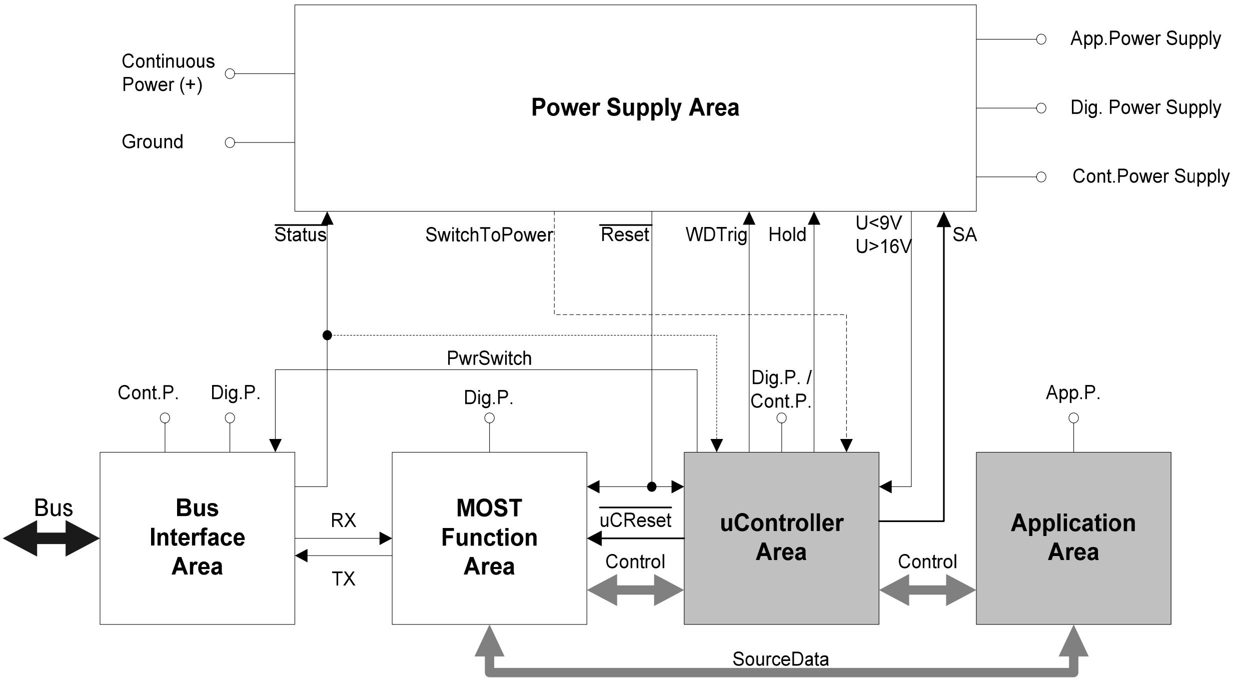


Figure 4-1: Example of the structure of a MOST device, the different functional areas and their interfaces

4.2 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 micro controller via I^{2}C (as Slave), SPI, MLB 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 bypass mode and switches to 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 clock signals generated by the Network Interface Controller are not valid. If these clock signals are used as device clock, this must be taken into consideration.

4.3 µC Area

The micro controller (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 that 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 that 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).

4.4 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, that is, 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).

4.5 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 4.6. 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 4-2 shows an example for the implementation of a Power Supply Area.

To meet these requirements, a MOST Network Interface Controller, micro controller (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 4-2, mainly consists of:

- Filter, unload-protection, EMI/EMC protection
- Micropower regulator (Cont.P.)
- SwitchToPower detector (optional)
- Power on logic
- Digital power supply (Dig.P.)
- Application power supply (App.P.)
- Bad power condition comparator
- Reset generator
- Watchdog timer

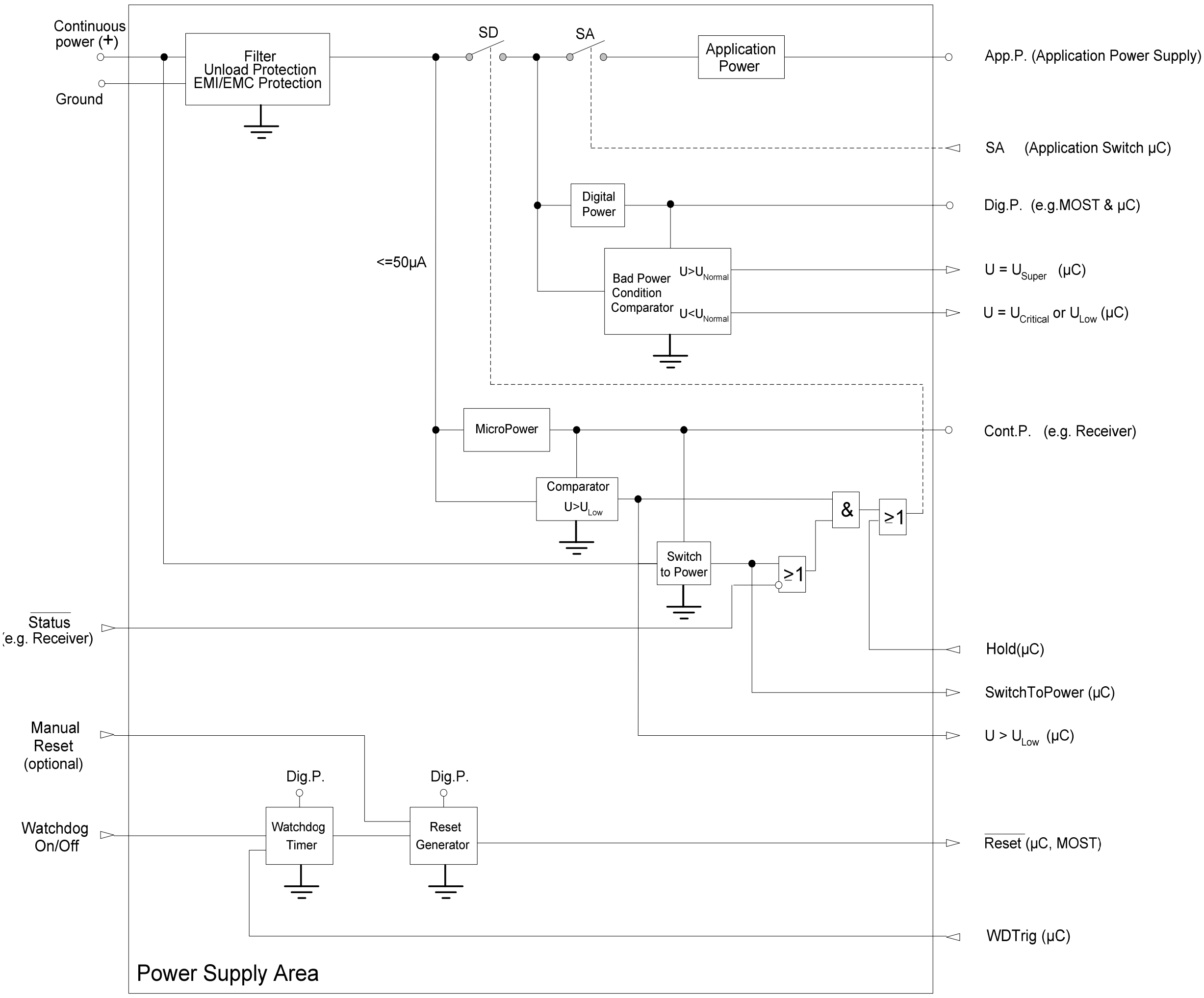


Figure 4-2: Block diagram of power supply area

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.

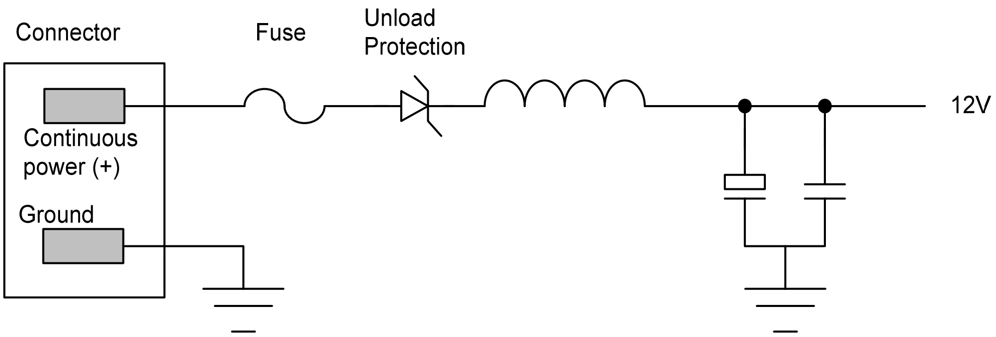


Figure 4-3: Input section of power supply area

The **Micropower Regulator** provides power supply for components 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 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.

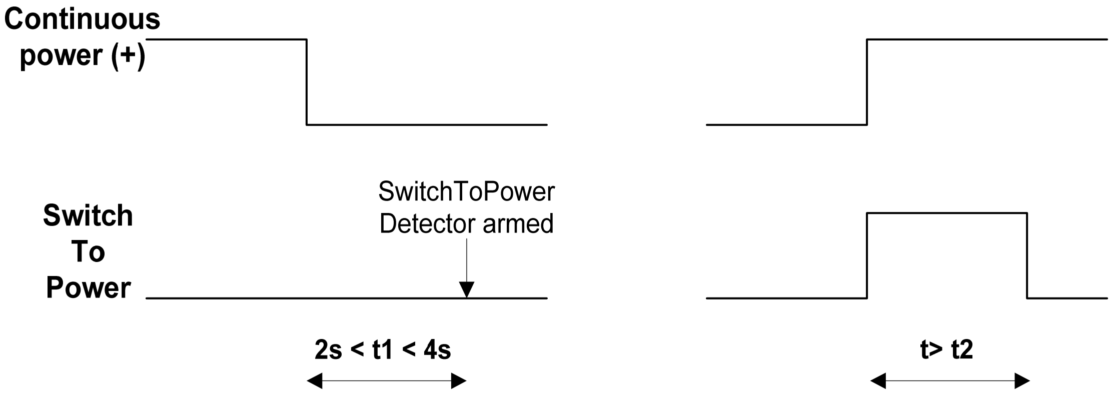


Figure 4-4: Timing of the output signal of the SwitchToPower detector depending on voltage at continuous power input

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 micro controller (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 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.

4.6 Voltage Levels

In general, a device in sleep mode must not wake the bus 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. A typical value for U_{Super} is 16V.

Critical voltage ($U_{Critical}$):

The device is in a safe operation state, which must be defined for each device individually. A typical value for $U_{Critical}$ is 9V. 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. A typical value for U_{Low} is 7V. 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 shut down its bus interface 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, for example, to enter Low Voltage as late as possible. Particularly when using switched power supplies, it can be possible to drop the Low Voltage threshold to a lower value.

2. Hysteresis ranges must be implemented to avoid oscillation!

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, for example, 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, for instance, by resetting the MOST Network Interface Controller, which will then close its bypass (except a device containing the TimingMaster). 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 3.2.5.5 on page 150.

5 Appendix A: Network Initialization

This Appendix contains some behavioral examples regarding Network Management. Requirements regarding Network Management behavior of the NetworkMaster and the Network Slaves can be found in section 3.2 and MOST Dynamic Specification.

5.1 NetworkMaster Section

This section contains scenarios of the behavior of the NetworkMaster during network initialization.

5.1.1 Flow of System Initialization Process by the NetworkMaster

The flow in Figure A-5-1 shows how the NetworkMaster initializes the system. Refer also to Figure A-5-2 for a flow of how the NetworkMaster performs the configuration requests from the Network Slaves during the system configuration.

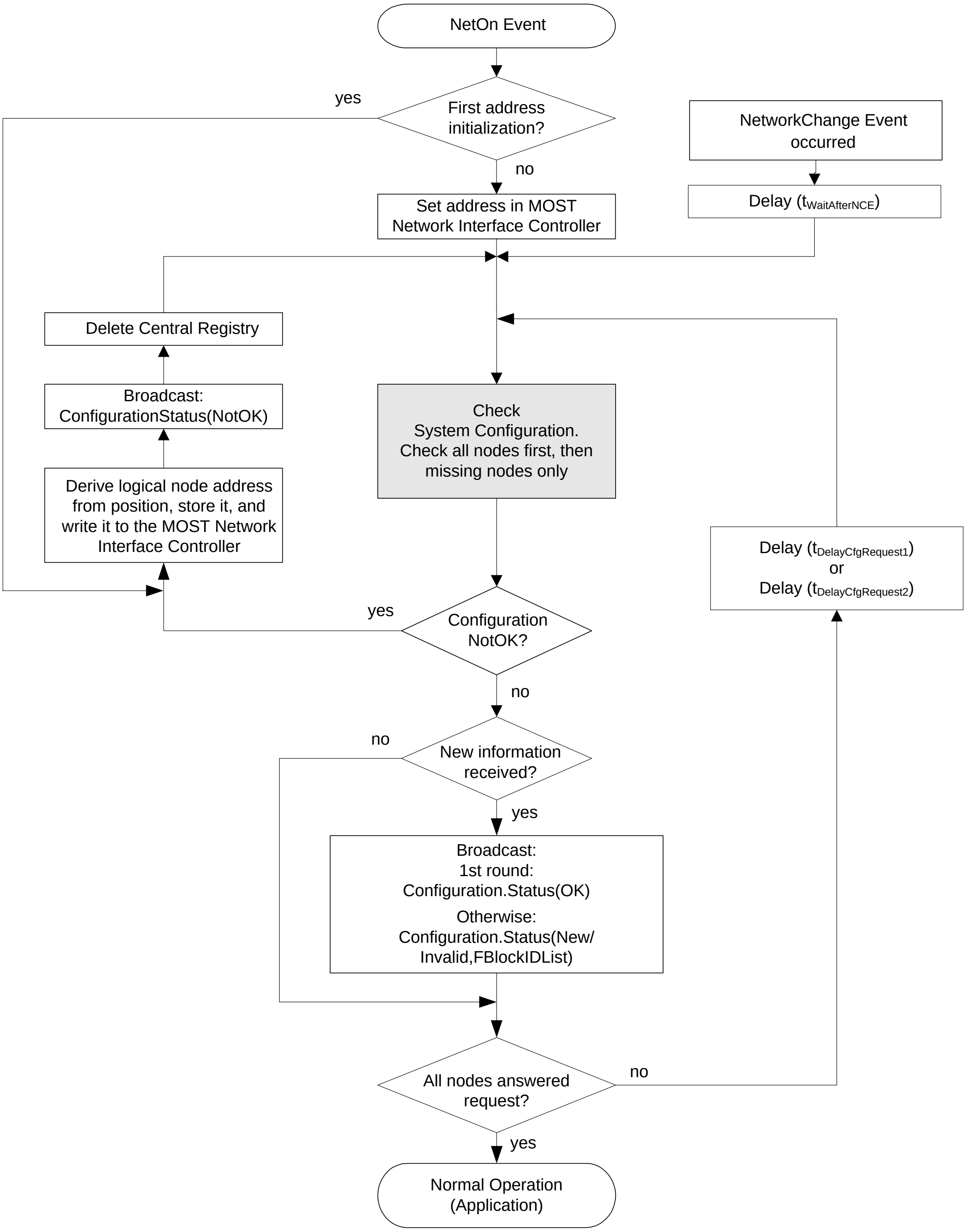

NetOn Event
First address initialization?
yes
no
NetworkChange Event occurred
Set address in MOST Network Interface Controller
Delay ($t_{WaitAfterNCE}$)
Delete Central Registry
Broadcast: ConfigurationStatus(NotOK)
Check System Configuration. Check all nodes first, then missing nodes only
Derive logical 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)
All nodes answered request?
no
yes
Normal Operation (Application)

Figure A-5-1: Flow of initialization on application level in a NetworkMaster

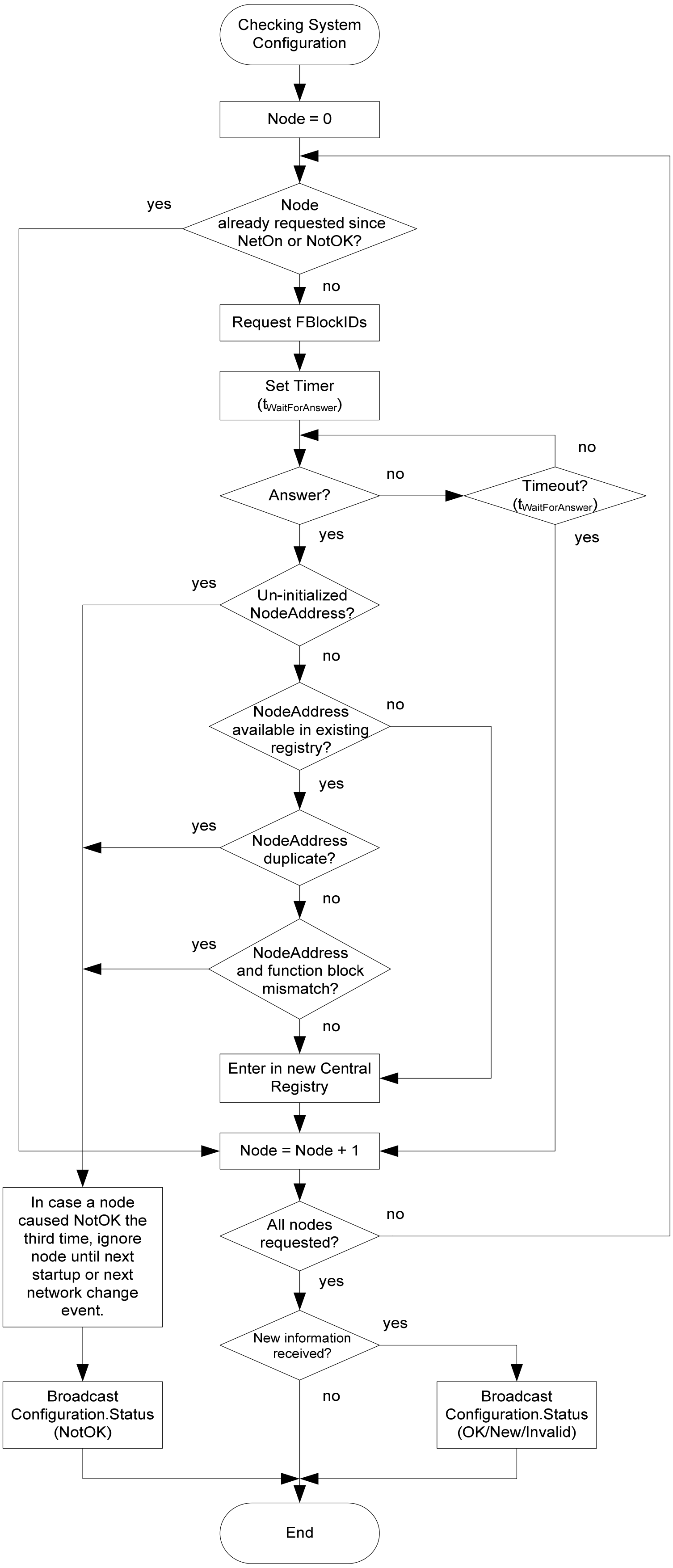

Checking System Configuration
Node = 0
Node already requested since NetOn or NotOK?
yes
no
Request FBlockIDs
Set Timer ($t_{WaitForAnswer}$)
no
Answer?
no
Timeout? ($t_{WaitForAnswer}$)
yes
yes
yes
Un-initialized NodeAddress?
no
NodeAddress available in existing registry?
no
yes
yes
NodeAddress duplicate?
no
yes
NodeAddress and function block mismatch?
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.

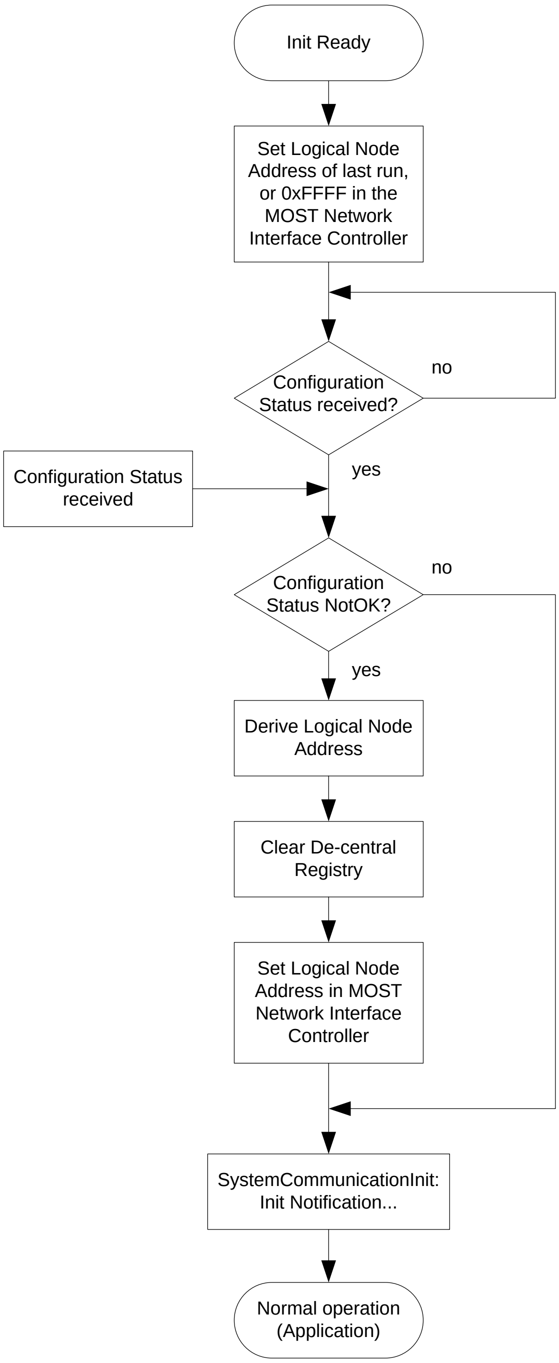

Init Ready
Set Logical Node Address of last run, or 0xFFFF in the MOST Network Interface Controller
Configuration Status received?
no
yes
Configuration Status received
Configuration Status NotOK?
no
yes
Derive Logical Node Address
Clear De-central Registry
Set Logical Node Address in MOST Network Interface Controller
SystemCommunicationInit: Init Notification...
Normal operation (Application)

Figure A-5-3: Flow of initialization on application level in a Network Slave

6 Appendix B: Typical Data Rates of Current Implementations (informative)

MOST25
At a system sample frequency of 44.1 kHz, 2,756 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 2,670. 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 arbitrate every third control frame. Therefore, a single device has a maximum message rate of 890 per second, or a net data rate of 135.28kBit/s.

MOST50
At a system sample frequency of 48.0 kHz, the gross data rate is 1.536 MBit/s, which results in up to 8000 messages per second for control messaging, depending on the message length.

In case all messages have the maximum size of payload (TelLen equals 12) the result is 5,333 messages per second. 8,000 Control Messages per second is the best case, if none of the messages transport additional payload (TelLen is 0).

When subtracting the data used for control and data securing, the net data rate (user data plus addressing) corresponds to up to 19 bytes per message:

Max: 810.62 kBit/s, in case of TelLen 12 (maximum payload)
Min: 448.00 kBit/s, in case of TelLen 0 (minimum payload)

A MOST device cannot immediately arbitrate again. Therefore, the maximum message rate is typicially reduced to 1/3. In case of maximum payload, a single device has a theoretical maximum message rate of 1,778 per second or a theoretical net data rate of 270.21 kBit/s. In case of minimum payload, a single device has a theoretical maximum message rate of 2,667 per second or a theoretical net data rate of 149.33 kBits/s.

7 Appendix C: List of Figures

8 Appendix D: List of Tables

INDEX

Document History (Previous Revisions)

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	0	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	3.1.4.1	Reserved device address 0x0FF0 as optional for debug purpose.

Change Ref.	Section	Changes
2V4_033	3.1.3.3.4 3.2.2.2 3.2.2.3 3.2.2.4 3.6 6	Removal of MOST transceiver register references.
2V4_034	3.2.2.2 3.2.5.1.2	Replaced t_{Master} and t_{Slave} with t_{Config}.
2V4_035	3.2.2.4	Replaced t_off by t_restart in Figure 3-14.
2V4_036	3.2.4	Changed description.
2V4_037	3.2.4.2 3.8	Added timer $t_{SlaveShutdown}$.
2V4_038	3.2.4.3.1	Increased description for “Request Stage”.
2V4_039	3.2.5.3	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 3.8	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	3.7.2.1	InstID changed to 1.
2V4_056	3.7.2.1	Extended parameter lists.
2V4_057	3.7.2.2	ResultAck changed to Result.
2V4_058	3.8	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.

Change Ref.	Section	Changes
2V3_005	General	Old chapter 3.1.5.2 removed.
2V3_006	General	Old chapter 3.1.5.7 removed
2V3_007	General	Old chapter 3.3 moved to 3.4 and some contents distributed to other chapters.
2V3_008	General	3.3.8 “Direct Access to OS8104” and 3.3.9 “Remote Control” were removed.
2V3_009	1	Chapter reworked, new document structure.
2V3_010	2.1.2	Connection Manager introduced.
2V3_011	2.1.4	MOSTSet Boundary removed
2V3_012	2.2.8	Is now MOST Network Service overview. Picture changed.
2V3_013	2.3.2.2	Connection Master not mandatory. FBlock Enhanced Testability added.
2V3_014	2.3.2.3	Description of InstIDs improved. Section on handling InstID of FBlock Enhanced Testability added.
2V3 015	2.3.2.3.2	Added Note: Wildcard must not be used for InstID assignment.
2V3 016	2.3.2.3.4	InstID NWM added
2V3_017	2.3.2.5.1	Error codes 0x08 and 0x09 deprecated. Application notified when segmentation error occurs. No ErrorAck on segmentation errors.
2V3_018	2.3.2.5.3	Added t_ProcessingDefault1, t_ProcessingDefault2, t_WaitForProcessing1, t_WaitForProcessing2 to text and pictures.
2V3_019	2.3.2.5.5	$t_{Property}$ introduced.
2V3_020	2.3.2.5.7	$t_{Property}$ introduced.
2V3_021	2.3.2.7	Reference to MOST High removed.
2V3_022	2.3.2.7.10	DAB Charsets added.
2V3_023	2.3.8	FBlock Enhanced Testability does not need to be listed.
2V3_024	2.3.11.2	Overview table added.
2V3_025	2.3.11.2.4.2	Reference to Layer 2 of NetService removed.
2V3_026	2.3.11.2.5	Function Class Sequence Property added.
2V3_027	2.3.11.3	Overview table added.
2V3_028	2.3.11.3.2	Function Class Sequence Method added.
2V3_029	2.3.12	Error handling extended.
2V3_030	3.1.1	Changed “Rx pin” to “Tx pin”
2V3_031	3.1.4.4.2	Remote Access removed. Transceiver register reference removed. Standalone mode removed. Remote GetSource added.
2V3_032	3.1.5.1	SAI removed. Table changed.
2V3_033	3.1.5.2	Standalone mode removed.
2V3_034	3.1.5.3	Transceiver specifics removed
2V3_035	3.1.5.4	Transceiver specifics removed. Time estimation removed.
2V3_036	3.1.5.8	Rewritten without transceiver registers.
2V3_037	3.2.2.2	As soon as the initialization of the MOST Network Interface Controller starts, the logical node address in the MOST Network Interface Controller has to be set to 0x0FFE.
2V3_038	3.2.2	Setting logical node address in MOST Network Interface Controller has been added to Figure 3-4, Figure 3-5, and Figure 3-6. Note that Figure 3-5 and have switched places from previous version.
2V3_039	3.2.2.4	t_Diag_Start and t_Diag_Restart added
2V3_040	3.2.2.4	Figure 3-10 and Figure 3-12, Diagnosis Normal Shut Down replaced by Diagnosis Ready.
2V3_041	3.2.4	Power Management section reworked. The Shutdown procedure has been divided into Network Shutdown and Device Shutdown. The Device Shutdown procedure is new.
2V3_042	3.2.5	Configuration.Status NotOk added as a case for securing synchronous data. Also changed so that sinks have to mute in case of an error. Not sources.
2V3_043	3.2.5.4	New definition of Network Change Event.
2V3_044	3.2.5.5	Note rewritten and maximum changed to 11 Bytes. NWM has InstID changed to 1
2V3_045	3.2.5.7	Failure of a Network Slave Device added.
2V3_046	3.2.5.8	Figure 3-17, Application may PowerOff in “Device Standby”. Voltage levels are no longer exact. Application removed from power states. Note added.
2V3_047	3.3	Network Management section is new. This section replaces section 3.2.3 and 3.3.5 of MOST Specification V2.2.
2V3 048	3.3.3.3.4	Introduced a new timer $t_{WaitAfterNetOn}$.
2V3 049	3.3.4.3.2	Deleted : The exception…(rest of paragraph)
2V3 050	3.3.4.3.8	Determination of System State clarified
2V3 051	3.4	This is old chapter 3.3
2V3 052	3.4.1	Address descriptions changed and Internal Node Communication Address added.
2V3 053	3.4.5	Section contains an example of Basics For Automatic Adding of Physical Address. This section is compiled from parts of section 3.3.5 of MOST Specification V2.2.
2V3 054	3.5	Chapter reworked and SourceConnect added. Mute was changed to SetGet.
2V3 055	3.5.1	NetServices routines removed.

Change Ref.	Section	Changes
2V3 056	3.5.2.1	Added remark that the SourceHandles function should only be used for debugging purposes.
2V3 057	3.5.2.2	Added that sources and sinks are numbered in ascending order starting from 1.
2V3 058	3.6.2.1	Ethernet Frames replaced by MAMAC Packets.
2V3 059	3.7	Boundary is now SetGet and NWM has InstID 1 in example.
2V3 060	3.8	Reworked. t_DeadlockPrev added. t_CleanChannels added and RemoteGetSource mentioned in Supervising Synchronous Connections.
2V3 061	3.8.1.4	Reworked supervising synchronous connections
2V3 062	3.9	New Timing Definition table.
2V3 063	4.1	Picture is only an example solution.
2V3 064	4.2.2	Table removed.
2V3 065	4.3	Standalone mode removed.
2V3 066	4.6	Absolute power values removed from the picture. Relative values introduced. SwitchToPower detector is optional.
2V3 067	4.7	Absolute power values replaced by relative values. Application changed to device.
2V3 068	Appendix A: Network Initialization	Added. Contains information from old chapter 3.4.

Changes MOST Specification 2V1-00 to MOST Specification 2V2-00

Change Ref.	Section	Changes
2V2_001	Bibliography	- Added [9] MAMAC Specification.
2V2_002	1	- Figure 1-1 updated with MAMAC. Function catalog has been split
2V2_003	2. 2. 2	- Events are only generated if requested.
2V2_004	2. 2. 4. 2	- SetResult changed to SetGet. Incrementing/decrementing added to the table.
2V2_005	2. 2. 10. 1	- Removed a table and the surrounding text.
2V2_006	2. 3. 2. 2	- Added Mandatory to Table2-2. Added the following function blocks to Table 2-2 and Table 2-3: Handsfree Processor: 0x28 DVD Video Player: 0x34 TMC Decoder: 0x53 Bluetooth: 0x54
2V2_007	2.3.2.3	- Changed default instance ID to 0x01. Removed two sentences that had to do with the old way the instance ids worked.
2V2_008	2.3.2.4	- Added NotificationCheck.
2V2_009	2.3.2.5.1	- Re-numbered the list correctly. Added text to Method Aborted
2V2_010	2.3.2.5.2	- Removed Result and Processing from the headline. Rephrased the text.
2V2_011	2.3.2.5.3	- Figure 2-16 and Figure 2-17 now use “yes” and “no”.
2V2_012	2.3.2.5.7	- Added information about the Get part of SetGet.
2V2_013	2.3.2.5.9	- Added Status, Error to the headline.
2V2_014	2.3.2.5.10 2.3.2.5.11	- Added Error/ErrorAck to the headlines. - Added references to Method Aborted.
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.

Change Ref.	Section	Changes
2V2_021	2.3.11.1	- Changed to a table. - Added Channel Type and Reserved bitfields. - Added that Unicode is not ASCII compatible - Added Table 2-9 and descriptions about the different modes that can be set through Channel Type. - Added Function Class Container (0x1B) - Changed 0x1A to BitSet
2V2_022	2.3.11.1.2	- Changed mils to miles in Table 2-10.
2V2_023	2.3.11.1.7	- Added Function Class Container.
2V2_024	2.3.11.2.3 2.3.11.2.4.2	- Changed <> to = in front of “0x01not allowed, no access to Tag”
2V2_025	2.3.11.2.4.3	- Clarified that parameters are not used in mode Top and Bottom. - Clarified what happens when an invalid position of the ArrayWindow is reached.
2V2_026	2.3.11.2.4.4	- Added Re-synchronization of ArrayWindows.
2V2_027	2.3.12	- Removed the requirement of three entries.
2V2_028	3.2.2.3	- Added text and Figure 3-7 to better explain how devices behave when unlocks occur.
2V2_029	3.2.2.4	- Changed timer from t_{Lock} to t_{Diag_Lock}
2V2_030	3.2.3.1	- Added chapter about Configuration Status Events
2V2_031	3.2.3.2	- Added information about t_{Bypass} which is set to 200ms. - Changed Figure 3-14 to include a new timer.
2V2_032	3.2.3.3	- Removed the requirement to store the Decentral Registry in buffered RAM.
2V2_033	3.2.5.1	- Added information about electrical wakeups.
2V2_034	3.2.6.3	- Added a sentence to explain that the behavior applies to all sinks.
2V2_035	3.2.6.4	- Added that the status message may not be sent before the NetworkMaster has asked the device.
2V2_036	3.2.6.6	- Removed the Power Save Mode and altered the text to fit the new structure. - Figure 3-21 was redrawn.
2V2_037	3.2.7	- Included a chapter about Over-Temperature Management.
2V2_038	3.3.1	- Changed the text about Node Position Address.
2V2_039	3.3.5.3	- Changed a “nein” to ”no” in Figure 3-23.
2V2_040	3.3.6	- Added a “yes” to Figure 3-24.
2V2_041	3.3.7.2	- Added a maximum length to segmented transfers.
2V2_042	3.4.1.1.2.1	- Changed the second parameter to RequiredChannels. - Added that allocation must not be done partially.
2V2_043	3.4.1.1.3	- Added “Handling of Handling of Double (De)/Allocate/(Dis)Connect Commands”
2V2_044	3.5.2.1	- Added TelID “B” for MAMAC48
2V2_045	3.5.3	- Removed chapter about MAMAC and included a short overview and a reference to [9].
2V2_046	3.7.1.2	- Point 5 was updated to RequiredChannels. Point 7 was removed.
2V2_047	3.8	- t_{Config} changed to 2000ms. - t_{Bypass} added. - $t_{WaitAfterNCE}$ changed to 200ms, and it is now a minimum - Added the Sentence “This time also applies to a shutdown after a slave-wakeup.” to $t_{ShutDown}$. - $t_{Restart}$ changed to 300 ms or MPR*15 ms. Added “The 300ms minimum applies to networks containing up to 20 devices. For larger networks the time can be calculated as follows: $t_{Restart}$ > MPR * 15ms”. - $t_{DelayCfgRequest}$ added to complement the Figure 3-14.
2V2_048	5	- This chapter was removed.

Changes MOST Specification 2V0-01 to MOST Specification 2V1-00

Change Ref.	Section	Changes
2V1_001	1	- Added paragraph introducing object oriented approach
2V1_002	2.3.2.2	- FBlockIDs “System Specific” and “Supplier Specific” added (WG-DA 2000-09-12)
2V1_003	2.3.2.4	- FktIDs “System Specific” and “Supplier Specific” added (WG-DA 2000-09-12) - Handling of proprietary Functions/ Function Blocks by controller added (WG-DA 2000-02-09)
2V1_004	2.3.2.2	- Speech output Device added (WG-DA 2000-09-12) - Speech Database Device added (WG-DA 2000-09-12) - Corrected FBlockIDs DAB Tuner (0x43) and TMCTuner (0x41) - FBlock Satellite Radio (0x44) added - FBlock HeadphoneAmplifier (0x23) added - FBlock AuxiliaryInput added (0x24) - FBlock MicrophoneInput added (0x26) - FBlock (0x51) "Telephone mobile" replaced by "Phonebook" - FBlock Router added (0x8) (WG-DA 2001-01-17)
2V1_005	2.3.2.5.1	- Specification of “Error Secondary node” revised - Specification of “Error Device Malfunction” added (WG-DA 2000-05-04) - Specification of “Segmentation Error” added (WG-DA 2000-09-12) - Hint to avoiding “infinite loops” added - "No error replies allowed in case of reception of broadcasted messages" added - Specification of “ Error Method Aborted” added (WG-DA 2000-11-22) - Added remark that methods in general should be aborted only by that application, which has started the method. - Code 0x05 and 0x06: Returning of the value of first incorrect parameter is optional (WG-DA 2001-01-17).
2V1_006	2.3.2.5	- Renamed StartAck -> StartResultAck (0x6) and adapted every occurrence in specification document. - Added AbortAck (0x7) - Added New StartAck (0x8)
2V1_007	2.3.2.5.4	- Added New StartAck (0x8)
2V1_008	2.3.2.5.11	- Added AbortAck (0x7)
2V1_009	2.3.2.6	- Maximum value for LENGTH changed to 65535
2V1_010	2.3.2.7	- Encoding of signed values added - Codes for ISO 8859/15 8 bit and UTF8 added - Maximum value for LENGTH changed to 65535 - Examples enhanced - Data type Boolean revised - Data type BitField added - Description of String enhanced (Null Strings)
2V1_011	2.3.2.5.3	- Flow chart “Flow for handling communication of methods (controller’s side)”. Error handling for “Timeout = YES” added - Changing of timeout (100ms) for "PROCESSING"
2V1_012	2.3.11.1.2	- Specification of NSteps extended - Units for Speed (m/s), Angle and Pixel added
2V1_013	2.3.11.1.4	- Interpretation of Increment and Decrement added
2V1_014	2.2.6	- Handling of dynamic changes of Function Interfaces through Notification added
2V1_015	General	- MMI replaced by HMI (Human Machine Interface)
2V1_016	3.9	- Description of Secondary Node added
2V1_017	3.2.6.8	- Section completely removed, due to an overlapping with the MOST Function Catalog

Change Ref.	Section	Changes
2V1_018	3.2	- Generally revised
	3.2.2	- Figure 3-3 “Diagnosis Normal Shutdown” changed to “Diagnosis Ready”
	3.2.2.1	- Table 3-6 changed
	3.2.3.2	- “Network Slave” removed, “Requesting System Configuration – Network Master” added
	3.2.3.3	- “Network Master” removed, “Requesting System Configuration – Network Slave” added
2V1_019	3.2.4	- Dynamic Behavior of Secondary Nodes added
2V1_020	3.2.6.4	- "Failure Of A Function Block" added
2V1_021	3.8	- Timeout $t_{Runtime}$ added - Timeout $t_{CfgStatus}$ changed - Timeout t_{Answer} changed - Timeout t_{Diag_Master} changed - Timeout t_{Diag_Slave} changed
2V1_022	2.3.12	- Error handling in case of property failure added - Notification of Function Interface (FI) added (WG-DA 2001-01-17) - Error handling added, in case of values in property not yet available during subscription (WG-DA 2001-01-17)
2V1_023	2.3.2.2	- Note about FBlockID 0xFF added
2V1_024	2.3.11.2.4.2	- Added parameters CurrentSize and AbsolutPosition to description of ArrayWindows - Added PositionTag, and descriptions for PositionTag and WindowSize (WG-DA 2001-01-17)
2V1_025	2.3.2.5.10	- Added remark that methods in general should be aborted only by that application, which has started the method.
2V1_026	2.3.2.5.11	- Function Class BoolField added - Function Class BitField added - Description of parameter "OPType" enhanced
2V1_027	2.3.11.1	- Start of Ring Break Diagnosis revised
2V1_028	3.2.5.1	- Note about wakeup methods added
2V1_029	4.6, 4.7	- Voltage levels and Implementation of Power Supply Area are no longer normative.
2V1_030	General	- Eithernet replaced by Ethernet
2V1_031	3.2.2.2	- Behavior of a waking Slave device (Figure 3-6)
2V1_032	3.5.3	- "MOST Asynchronous Medium Access Control (MAMAC)" added
2V1_033	3.4.3.3	- Equation for delay compensation revised ($T_{Source} < T_{Node}$)
2V1_034	4.2.4	- Hint Added. Description of pig tail is only one of the possible implementations.
2V1_035	3.4.1.1.2.1	- Method SourceActivity added
2V1_036	3.2.6	- General handling of errors. Synch. connections are removed in case of Fatal Errors.

Changes MOST Specification 2V0-00 to MOST Specification 2V0-01

Change Ref.	Section	Changes
2V01_001	General	Document no longer specified as “Confidential”; Legal Notice inserted.

Changes MOST Specification 1V0 to MOST Specification 2V0

Change Ref.	Section	Changes
2V0_001	3.3.1	Equation modified; Startup address 0xFFFF
2V0_002	2.1.2/ 2.2.5	Section 2.2.5 moved to 2.2.2
2V0_003	2.2.1	NetBlock “functions related to the entire device.”
2V0_004	2.3.2.2	Table 2-5: Proprietary FBlockIDs 0xF0..0xFE
2V0_005	2.3.2.3	Completely revised
2V0_006	2.3.2.4	Minor modification
2V0_007	2.3.2.5	Completely revised
2V0_008	2.3.2.6	Completely revised
2V0_009	2.3.2.7	Boolfield introduced; Definition of STRING expanded, Examples for Exponent, Step and Unit
2V0_010	2.3.5	Minor modification
2V0_011	2.3.6	Distinguishing Properties and Methods; Communication with routing revised
2V0_012	2.3.10	Transmitting function interfaces. Introduced.
2V0_013	2.3.11	Function Classes (completely revised)
2V0_014	2.3.12	Notification for array properties; Notification re-build at system start
2V0_015	3.2.2.2	Error_t_slave replaced by Error_NSInit_Timeout
2V0_016	3.2.2.3	Completely revised
2V0_017	3.2.2.4	Completely revised
2V0_018	3.2.3.2	Completely revised
2V0_019	3.2.3.2	Completely revised
2V0_020	3.2.5.1	Completely revised
2V0_021	3.2.6	General rules added
2V0_022	3.2.6.1	Completely revised
2V0_023	3.2.6	Completely revised
2V0_024	3.3.5.3	- Table 3-14; - sample for receiving logical node address; - section below Table 3-15
2V0_025	3.3.7.2	TelIDs for MOST High Protocol removed
2V0_026	3.3.8.1	Figure 3-25; Set STX bit added
2V0_027	3.4.1.1.1	Replaced “.0.” by “.Pos.”
2V0_028	3.4.1.1.2	Completely revised
2V0_029	3.4.3.3	Equations
2V0_030	3.5.2.1	TelID and TelLen changed; One ID reserved for Ethernet frames
2V0_031	3.7.1.1	Revised (OPTypes)
2V0_032	3.7.1.2	Revised (OPTypes)
2V0_033	3.7.1.3	Revised (OPTypes)
2V0_034	3.1.4.2	Table 3-1
2V0_035	3.1.4.2.2	Revised
2V0_036	3.1.4.3.1	Handling of Isochronous data removed

Change Ref.	Section	Changes
2V0_037	3.1.4.3.6	Table 3-2; Table 3-3 added, Handling of Isochronous data removed
2V0_038	3.1.4.4.2	Completely revised
2V0_039	4.1	Figure 4-1
2V0_040	4.2.1	Completely revised
2V0_041	4.2.2	Revised
2V0_042	4.2.4	Completely revised
2V0_043	4.3	Revised
2V0_044	4.5	Completely revised
2V0_045	4.6	Completely revised
2V0_046	4.7	Completely revised
2V0_047	---	General changes in Structure: - Chapter 2.1 removed, contents included within 2.2.9 - Detailed descriptions of Control Channel (2.2) moved to 3.3 - Introduction of CMS/ AMS moved to 3.3.7 - Chapters 2.6 up to 2.12 moved to 3.2 up to 3.8 - Chapter 2.5 and 2.13 moved to Chapter 5

Notes:

Notes:

Notes: