

MOST

Media Oriented Systems Transport

Multimedia and Control
Networking Technology

MOST Specification

Rev 2.0

12/1999

Version 2.0-01



Legal Notice

COPYRIGHT

© Copyright 1999 - 2000 MOST Cooperation. All rights reserved.

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
P. O. Box 4327
D-76028 Karlsruhe
Germany

Tel: (+49) (0) 721 966 50 00

Fax: (+49) (0) 721 966 50 01

E-mail: contact@mostcooperation.com

Web: www.mostcooperation.com



© Copyright 1999 - 2000 MOST Cooperation
All rights reserved

MOST is a registered trademark

Contents

1	INTRODUCTION	11
2	APPLICATION SECTION	12
2.1	Overview of Data Channels	12
2.1.1	Control Channel	12
2.1.2	Synchronous Channel	12
2.1.3	Asynchronous Channel	13
2.1.4	Managing Synch./Async. Bandwidth	13
2.2	Logical Device Model	14
2.2.1	Function Block	14
2.2.1.1	Slave, Controller, MMI	15
2.2.1.2	First Introduction to MOST Functions	15
2.2.2	Functions	16
2.2.3	Methods	16
2.2.4	Properties	17
2.2.4.1	Setting a Property	17
2.2.4.2	Reading a Property	18
2.2.5	Events	18
2.2.6	Function Interfaces	19
2.2.7	Definition Example	20
2.2.8	MOST Data Flow Model	22
2.2.9	MOST System Services	23
2.2.10	Delegation, Heredity, Device Hierarchy	23
2.2.10.1	Delegation	23
2.2.10.2	Heredity of Functions	25
2.2.10.3	Deriving Devices/Device Hierarchy	26
2.3	Protocols	29
2.3.1	Protocol Basics	29
2.3.2	Structure of MOST Protocols	29
2.3.2.1	DeviceID	29
2.3.2.2	FBlockID	30
2.3.2.3	InstID	31
2.3.2.4	FktID	32
2.3.2.5	OPType	33
2.3.2.5.1	Error	34
2.3.2.5.2	Start, Result, Processing, Error	37
2.3.2.5.3	StartResult, Result, Processing, Error	37
2.3.2.5.4	StartAck, ProcessingAck, ResultAck, and ErrorAck	39
2.3.2.5.5	Get, Status, Error	39
2.3.2.5.6	Set, Status, Error	39
2.3.2.5.7	SetGet, Status, Error	39
2.3.2.5.8	GetInterface, Interface, Error	40
2.3.2.5.9	Increment And Decrement	40
2.3.2.5.10	Abort	40
2.3.2.6	Length	41
2.3.2.7	Data	41
2.3.3	Function Formats in Documentation	43
2.3.4	Protocol Catalogs	43
2.3.5	Application Functions on MOST Network (Introduction)	44
2.3.6	Controller/Slave Communication	47
2.3.6.1	Communication With Properties Using Shadows	47
2.3.6.2	Communication With Methods	52
2.3.6.2.1	Standard Case	52
2.3.6.2.2	Special Case Using Routing	53
2.3.7	Seeking Communication Partner	55
2.3.8	Requesting Function Block Information from a Device	55
2.3.9	Requesting Functions from a Function Block	56

2.3.10	Transmitting The Function Interface	57
2.3.10.1	Principle	57
2.3.10.2	Realization Of The Ability To Extract The Function Interface	57
2.3.11	Function Classes	58
2.3.11.1	Properties With A Single Variable	58
2.3.11.1.1	Function Class Switch	60
2.3.11.1.2	Function Class Number	61
2.3.11.1.3	Function Class Text	63
2.3.11.1.4	Function Class Enumeration	64
2.3.11.2	Properties with Multiple Variables	65
2.3.11.2.1	Function Class Record	66
2.3.11.2.2	Function Class Array	68
2.3.11.2.3	Function Class Dynamic Array	71
2.3.11.2.4	Function Class LongArray	73
2.3.11.3	Function Class For Methods	80
2.3.12	Handling Message Notification	81
3	NETWORK SECTION	84
3.1	MOSTTransceiver and its Internal Services	84
3.1.1	Electrical Bypass (All Bypass)	84
3.1.2	Source Data Bypass	84
3.1.3	Master/Slave, Active and Passive Components	84
3.1.4	Data Transport	85
3.1.4.1	Blocks	85
3.1.4.2	Frames	85
3.1.4.2.1	Preamble	87
3.1.4.2.2	Boundary Descriptor	87
3.1.4.2.3	MOST System Control Bits	87
3.1.4.3	Source Data	88
3.1.4.3.1	Definition of Control Data and Source Data	88
3.1.4.3.2	Differentiating Synchronous and Asynchronous Data	88
3.1.4.3.3	Source Data Interface	88
3.1.4.3.4	Transparent Channels	88
3.1.4.3.5	Synchronous Area	89
3.1.4.3.6	Asynchronous (Packet Data) Area	89
3.1.4.4	Control Data	91
3.1.4.4.1	Control Data Interface	91
3.1.4.4.2	Description	91
3.1.5	Internal Services	93
3.1.5.1	Addressing	93
3.1.5.2	Address Initialization (SAI)	93
3.1.5.3	Support at System Startup	94
3.1.5.4	Delay Recognition	94
3.1.5.5	Remote-Access	94
3.1.5.6	Automatic Channel Allocation	94
3.1.5.7	Power Management	95
3.1.5.8	Detection of Unused Channels	95
3.2	Dynamic Behavior of a Device	96
3.2.1	Overview	96
3.2.2	NetInterface	98
3.2.2.1	NetInterfacePowerOff	99
3.2.2.2	NetInterfaceInit	99
3.2.2.3	NetInterfaceNormalOperation	103
3.2.2.4	NetInterface Ring Break Diagnosis	105
3.2.3	Initialization on Application Level	111
3.2.3.1	Network Slave	111
3.2.3.2	Network Master	113
3.2.4	Power Management	117
3.2.4.1	General Procedure	117
3.2.4.2	Functions and Important Operations	120

3.2.5	Error Management	121
3.2.5.1	Handling of Light Off	121
3.2.5.2	Fatal Error	122
3.2.5.2.1	Waking	122
3.2.5.2.2	Operation	122
3.2.5.3	Unlock	123
3.2.5.4	NetworkChange Event	124
3.2.5.5	Low Voltage	124
3.2.5.6	"Hanging" of an Application	126
3.2.5.7	Diagnosis	126
3.3	Accessing Control Channel	127
3.3.1	Addressing	127
3.3.2	Assigning Priority Levels	129
3.3.3	Low Level Retries	129
3.3.4	High Level Retries	129
3.3.5	MOSTNetServices (Application Socket)	130
3.3.5.1	Basics for Automatic Adding of Physical Address	130
3.3.5.2	De-Central Registry	130
3.3.5.3	Central Registry	131
3.3.6	Handling Overload in a Message Sink	134
3.3.7	MOSTNetServices (Basic Layer)	135
3.3.7.1	Control Message Service	135
3.3.7.2	Application Message Service (AMS) And Application Protocols	135
3.3.8	Direct Access to OS8104	137
3.3.8.1	Sending Messages	137
3.3.8.2	Receiving Messages	138
3.3.8.3	Acknowledgement and Data Security	138
3.3.9	Remote Control	139
3.3.9.1	Remote Read Message	139
3.3.9.2	Remote Write Message	140
3.4	Handling Synchronous Data	141
3.4.1	MOSTNetServices (Application Socket)	141
3.4.1.1	Basic Functions on Application Level	142
3.4.1.1.1	NetBlock	142
3.4.1.1.2	Function Block	143
3.4.2	MOSTNetServices (Basic Layer)	149
3.4.3	Direct Access to OS8104	149
3.4.3.1	Serial Interface	149
3.4.3.2	Parallel Interface	149
3.4.3.3	Compensating Network Delay	149
3.5	Handling Asynchronous (Packet) Data	150
3.5.1	Direct Access to OS8104	150
3.5.1.1	Priorities	150
3.5.2	MOSTNetServices	151
3.5.2.1	Securing data	151
3.6	Controlling Synchronous/Asynchronous Bandwidth	153
3.7	Connections	154
3.7.1	Synchronous Connections	154
3.7.1.1	Administering (Connection Master)	154
3.7.1.2	Establishing Synchronous Connections	156
3.7.1.3	Removing Synchronous Connections	158
3.7.1.4	Supervising Synchronous Connections	158
3.8	Timeouts	159
4	HARDWARE SECTION	161
4.1	Basic HW Concept	161

4.2	Optical Interface Area.....	162
4.2.1	Overview	162
4.2.2	Optical Power Budget	164
4.2.3	POF	164
4.2.4	Connection Systems (Pig Tail).....	165
4.3	MOST Function Area.....	166
4.4	µC Area	166
4.5	Application Area	167
4.6	Power Supply Area.....	167
4.7	Voltage Levels	172
5	TOOLS	174
5.1	OptoLyzer4MOST [®]	174
5.2	MOSTRapidControl	175
5.3	OptoLyzer4MOST [®] Professional.....	176
5.3.1	Introduction.....	176
5.3.2	Architecture	176
5.3.3	OptoLyzer Control	176
5.3.4	Bus Analysis.....	177
5.3.5	MOST Simulation Interface	177
6	APPENDIX A: INDEX OF FIGURES	179
7	APPENDIX B: INDEX OF TABLES	181
8	APPENDIX C: INDEX.....	182

Bibliography

Number	Document
[1]	MOST Specification Framework
[2]	MOST Specification
[3]	MOSTHighProtocol Specification
[4]	MOSTNetServices "Basic Layer"; User Manual and Specification
[5]	MOSTNetServices "Application Socket"; User Manual and Specification
[6]	FOT Datasheet
[7]	MOSTTransceiver Datasheet
[8]	MOSTFunctionCatalog

Document History

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

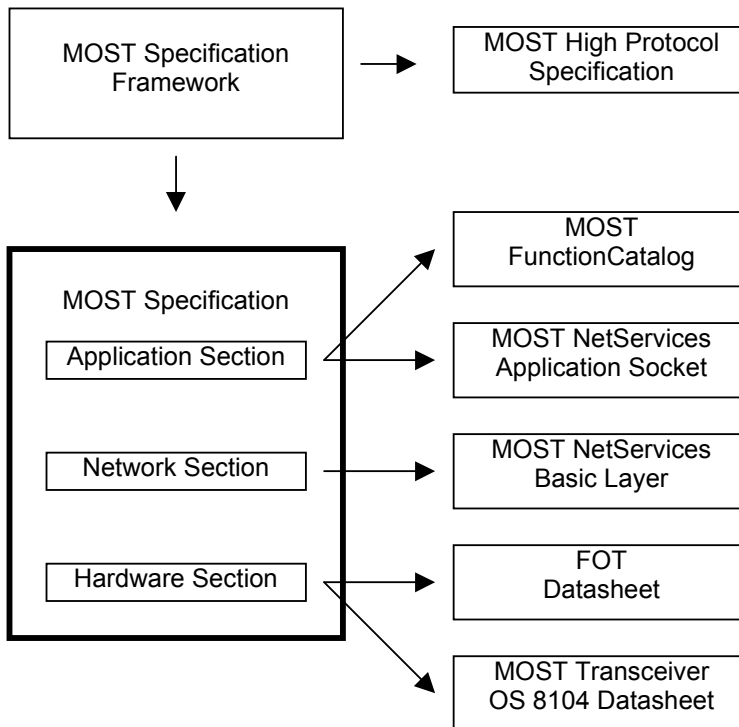
Change Ref.	Section	Changes
2V0_001	3.3.1	Equation modified; Startup address 0xFFFF
2V0_002	2.1.2/ 2.2.5	Section 2.2.5 moved to 2.2.2
2V0_003	2.2.1	NetBlock "functions related to the entire device."
2V0_004	2.3.2.2	Table 2-5: Proprietary FBlockIDs 0xF0..0xFE
2V0_005	2.3.2.3	Completely revised
2V0_006	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	Bool-field 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	3.2.3.1	Completely revised
2V0_019	3.2.3.2	Completely revised
2V0_020	3.2.4.1	Completely revised
2V0_021	3.2.5	General rules added
2V0_022	3.2.5.1	Completely revised
2V0_023	3.2.5	Completely revised
2V0_024	3.3.5.3	- Table 3-15; - sample for receiving logical node address; - section below Table 3-16
2V0_025	3.3.7.2	TelIDs for MOST High Protocol removed
2V0_026	3.3.8.1	Figure 3-22; Set STX bit added
2V0_027	3.4.1.1.1	Replaced ".0." by ".Pos."
2V0_028	3.4.1.1.2	Completely revised

Change Ref.	Section	Changes
2V0_029	3.4.3.3	Equations
2V0_030	3.5.2.1	TelID and TelLen changed; One ID reserved for Ethernet frames
2V0_031	3.7.1.1	Revised (OPTypes)
2V0_032	3.7.1.2	Revised (OPTypes)
2V0_033	3.7.1.3	Revised (OPTypes)
2V0_034	3.1.4.2	Table 3-1
2V0_035	3.1.4.2.2	Revised
2V0_036	3.1.4.3.1	Handling of Isochronous data removed
2V0_037	3.1.4.3.6	Table 3-2; Table 3-3 added, Handling of Isochronous data removed
2V0_038	3.1.4.4.2	Completely revised
2V0_039	4.1	Figure 4-1
2V0_040	4.2.1	Completely revised
2V0_041	4.2.2	Revised
2V0_042	4.2.4	Completely revised
2V0_043	4.3	Revised
2V0_044	4.5	Completely revised
2V0_045	4.6	Completely revised
2V0_046	4.7	Completely revised
2V0_047	---	General changes in Structure: - Chapter 2.1 removed, contents included within 2.2.9 - Detailed descriptions of Control Channel (2.2) moved to 3.3 - Introduction of CMS/ AMS moved to 3.3.7 - Chapters 2.6 up to 2.12 moved to 3.2 up to 3.8 - Chapter 2.5 and 2.13 moved to Chapter 5

1 Introduction

This document is part of the specification documentation of the MOST (Media Oriented Systems Transport) system. It contains the detailed specification of the application layer, the network layer, and the MOST hardware. For an overview of the MOST system, please refer to [1].

This is the structural overview of the MOST specification documentation:



More detailed information about individual items can be found in the associated documents:

- MOSTNetServices “Application Socket”
- MOSTNetServices “Basic Layer”
- FOT Datasheet
- MOSTTransceiver Datasheet
- MOSTFunctionCatalog

This document specifies the MOST system services, which are needed to develop MOSTDevices, i.e. hardware using MOST technology.

2 Application Section

2.1 Overview of Data Channels

2.1.1 Control Channel

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

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

In contrast to that, the asynchronous area is specified for transmissions requiring high bandwidth in a burst-like manner. Connections on the asynchronous channel are administered via the control channel.

2.1.2 Synchronous Channel

Continuous data streams that demand high bandwidth are transported over the synchronous channels. The connections are administered dynamically via the control channel. No bandwidth is reserved for special applications. Although synchronous connections can be built directly by source and sink nodes, it is recommended that available bandwidth be administered in a central manner, particularly in larger networks. Administration of the synchronous resources starts with the respective routines of the NetServices, via basic routines on the application level, and continues up to the administration of connections by a higher control instance.

So administration of the synchronous resources would be implemented in the connection master function block. Since the connection master must check to see if the connection already exists before the connection can be built, all requests for establishing connections must be directed to the connection master.

2.1.3 Asynchronous Channel

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

2.1.4 Managing Synch./Async. Bandwidth

On the MOSTNetwork there are 60 bytes available for synchronous and asynchronous data transfer. It is possible to divide up these resources between synchronous and asynchronous channels by means of a boundary descriptor. The boundary descriptor can be modified either by direct access to the respective register in a MOSTTransceiver, or by the MOSTNetServices (using MOSTSetBoundary).

The position of the boundary descriptor depends on the requirements of the system and can be changed dynamically. Supervision and changing of the bandwidth or position of the boundary is done in the timing master. The timing master is responsible for forwarding the information about the boundary's position to all nodes in the network. This task is handled automatically on the chip level. After having changed the boundary descriptor, the synchronous connections must be re-built.

2.2 Logical Device Model

The following sections describe different kinds of devices. A MOSTDevice contains at least one MOSTTransceiver and therefore is accessible via a MOSTNetwork. MOSTDevice means a physical unit which can be connected to a MOSTNetwork.

2.2.1 Function Block

On the application level, a MOSTDevice contains multiple components, which are called function blocks, e.g., tuner, amplifier, or CD player. It is possible that there are multiple function blocks in a single MOSTDevice, such as a tuner and an amplifier combined in one case and connected to the MOSTNetwork via a common MOSTTransceiver. In addition to the function blocks which represent applications, each MOSTDevice has a special function block called the NetBlock. The NetBlock provides functions related to the entire device. Between the function blocks and the MOSTTransceiver, NetServices form an intermediate layer providing routines to simplify the handling of the MOSTTransceiver.

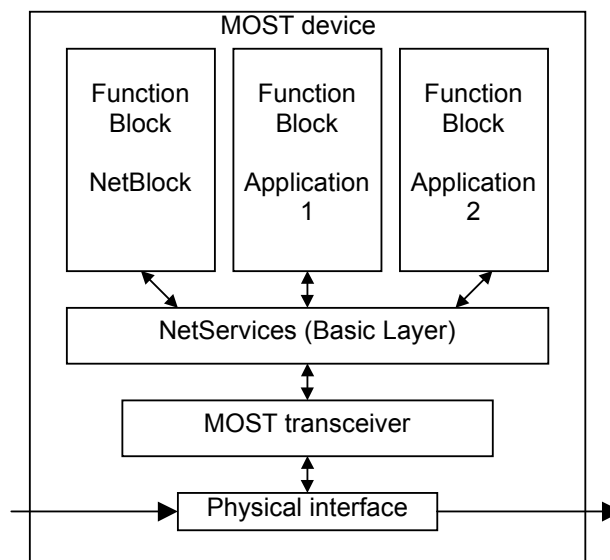


Figure 2-1: Model of a MOSTDevice

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

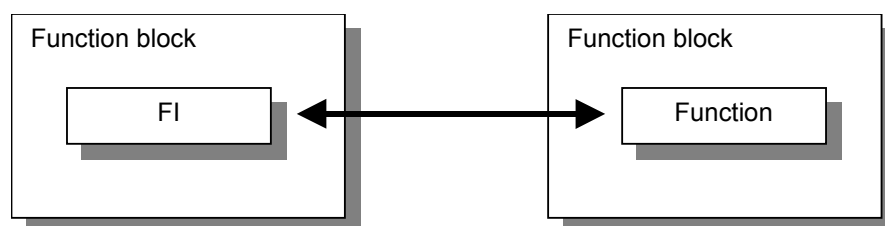


Figure 2-2: Communication with a function via its function interface (FI)

2.2.1.1 Slave, Controller, MMI

Function blocks that are always controlled are called slaves. Those function blocks that have an interface to the user are called MMIs. In addition to that, there are function blocks that combine multiple functions of different function blocks to higher functions?. They control, but may also be controlled. Those function blocks are called controllers.

A clear separation between MMI, controller, or slave cannot always be applied to devices, but in most cases a device can be classified with respect to its primary function.

2.2.1.2 First Introduction to MOST Functions

This section gives a brief introduction to the structure of MOST functions, as this knowledge is necessary to understand the following examples. Chapter 2.3 on page 29 explains the structure of MOST functions in more detail.

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

FBlockID . FktID

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

FBlockID . FktID . OPType (Data)

2.2.2 Functions

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

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

In addition to that, “events” can be defined. Events result from properties, if the properties report changes by themselves (Notification).

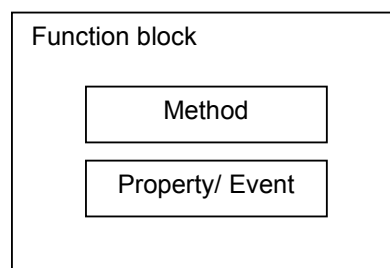


Figure 2-3: Structure of a function block consisting of functions classifiable as methods, properties and events.

2.2.3 Methods

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

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

After finishing the process, the controlled function block should report execution to the controlling function block (in a control device). This report might contain results of the process, for example, a frequency found by the tuner. If a process runs for a long time, it might be useful to return intermediate results before finishing, such as informing the controlling function block about the successful start of the process.

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

Controller	Slave
Start of a Method with Parameters (<i>Start/StartResult</i>)	Error with cause for error (<i>Error</i>) Execution report with results (<i>Result</i>) Intermediate result (<i>Processing</i>)

The respective MOST functions needed for this messaging are described in depth in chapter 2.3 on page 29.

2.2.4 Properties

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

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

2.2.4.1 Setting a Property

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

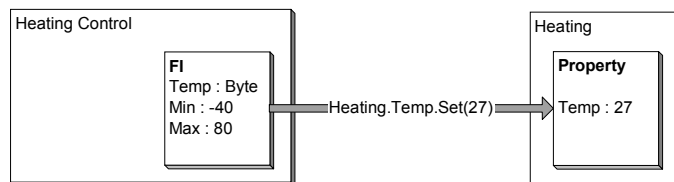


Figure 2-4: Setting a property (Temperature setting of a heating)

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

2.2.4.2 Reading a Property

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

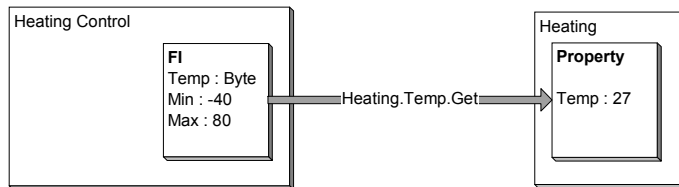


Figure 2-5: Reading a property (Temperature setting of a heating)

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

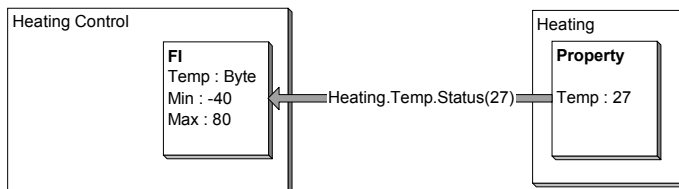


Figure 2-6: Status report of property temperature setting

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

Controller	Slave
Setting a property (<i>Set/SetResult</i>)	Status of property (<i>Status</i>)
Reading a property (<i>Get</i>)	Error message with cause of error (<i>Error</i>)

The MOST functions needed for this messaging are described in depth in chapter 2.3 on page 29.

2.2.5 Events

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

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

Events can be used to notify reaching of limits, or the change of measured values in function blocks (e.g., the play time of a CD player has changed), or in the MMI (e.g., reception of a new value of mileage received via a CAN gateway). Events are sent only to those function blocks that have requested it by an entry in the notification matrix (refer to chapter 2.3.12 on page 81). The respective data should be transmitted in the same message with events.

2.2.6 Function Interfaces

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

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

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

The contents of the FIs are usually known during implementation of a device (well known functions). It is also possible that FIs are transported on the bus during runtime, making it possible to dynamically reconfigure an MMI. In this way, even functions that did not exist during the development of an MMI can be made available.

Example:

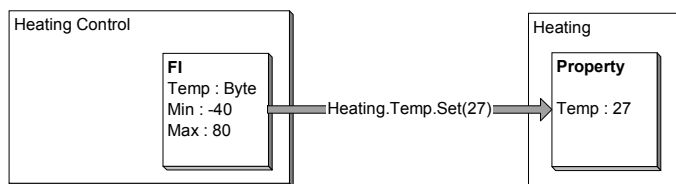


Figure 2-7: Example for a function interface (FI)

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

2.2.7 Definition Example

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

<pre>MyTuner = Device Tuner : TTuner; NetBlock : TNetBlock; end;</pre>	<pre>Device { TTuner Tuner; TNetBlock NetBlock; } MyTuner</pre>	<p>(Pascal Syntax)</p>	<p>(C Syntax)</p>
--	---	------------------------	-------------------

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

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

<pre>TTuner = Object pStation : TStation; eTraffic : TTraffic; pSensitivity: TSensitivity; mSearch : TSearch; end;</pre>	<pre>Object { TStation pStation; TTraffic eTraffic; TSensitivity pSensitivity; TSearch mSearch; } TTuner</pre>
--	--

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

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

Now property pStation will be defined as follows:

<pre>TStation = Property Frequency : Long; TP : Bool; Quality : Byte; end;</pre>	<pre>Property { long Frequency; Bool TP; Byte Quality; } TStation;</pre>
--	--

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

Now method mSearch will be defined:

<pre>TSearch = Method Up : Bool; Start : Long; end;</pre>	<pre>Method { Bool Up; Long Start; } Tsearch;</pre>
---	---

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
    TA : Bool;
end;

Event
{
    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
```

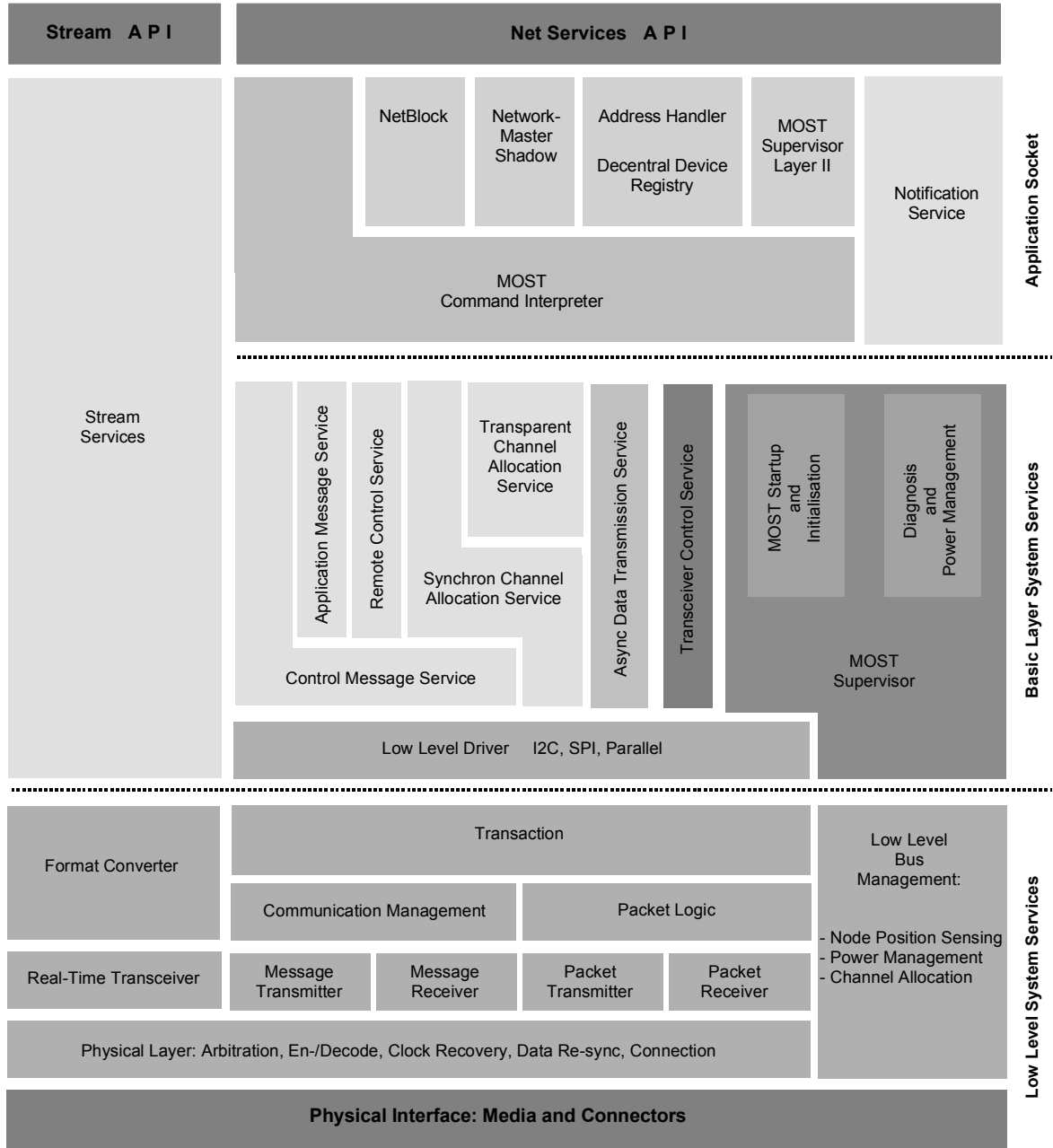
Here is the example for the interface description of parameter Frequency.

```
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 Data Flow Model

Block diagram of the MOST System Services



2.2.9 MOST System Services

The MOST System Services provide all the basic functionality to operate a MOST system. They are designed to offer the maximum flexibility with the greatest ease of use and consist of :

- Application socket
- Basic layer system services
- Low level system services
- Stream services

The low level system services, except for the physical interface, are implemented in the MOSTTransceiver. Stream services, basic layer system services, and application socket are implemented in the NetServices. For more detailed information please refer to [4] and [5].

The MOST system services offer a wide variety of functions for implementing applications. Some functions or properties are mandatory for a MOSTDevice. MOSTDevices should be able to handle control tasks in a peer to peer manner. To provide flexibility in control tasks, MOSTDevices must be able to work in an environment with multiple masters.

MOST system services provide a basic framework for a MOSTDevice.

2.2.10 Delegation, Heredity, Device Hierarchy

2.2.10.1 Delegation

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

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

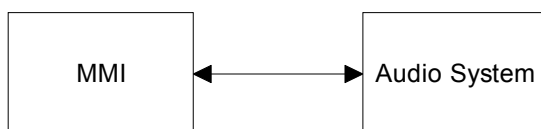


Figure 2-8: Ideal audio system

Real audio systems generally look like this:

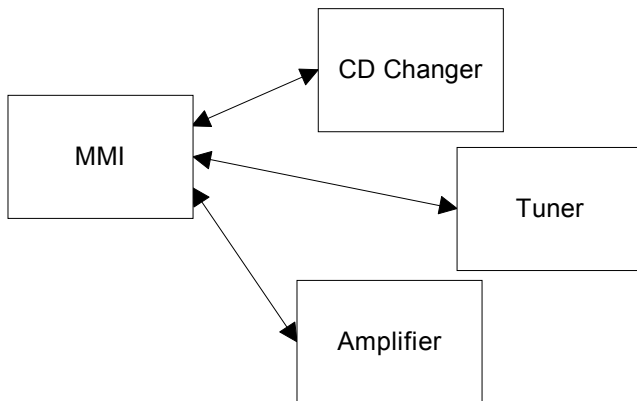


Figure 2-9: Real audio system

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

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

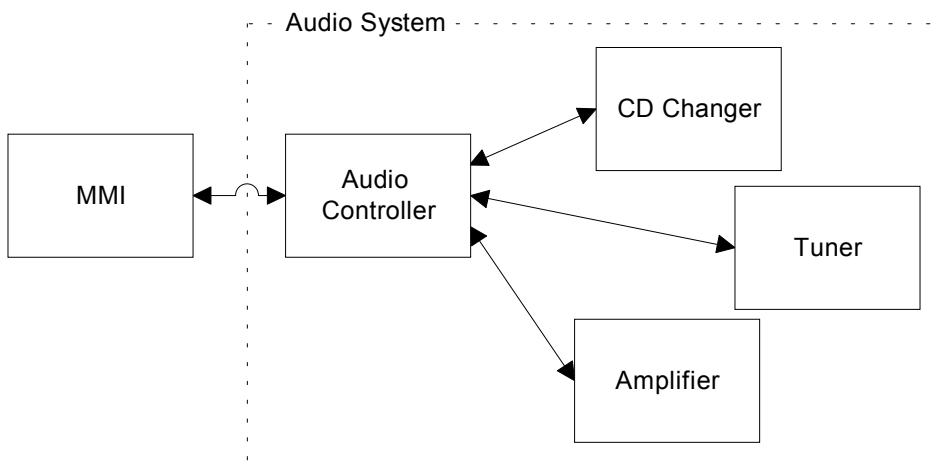


Figure 2-10: Delegation of functions of all audio components to one audio controller

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

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

Before the announcement:

Switch CD to pause, switch amplifier to tuner channels (eventually send special values for volume and sound).

After announcement:

CD back to play, amplifier back to CD, restore volume and sound.

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

The mechanism described above creates a problem during system initialization. If the MMI looks for the function TA in the network, it would be offered by two different devices (tuner and audio controller). One possible solution to this problem is to subdivide the function blocks in hierarchy layers (ranking). This ranking must be coordinated with the entire system, to be unambiguous system-wide. A sample ranking is shown in the following table:

Class	Ranking
Slave	0
Controller	1
System Controller	2
MMI	3

Table 2-1: Ranking of device classes

To every function block, a class (with the respective ranking) is assigned during design. If a device is asked for its function blocks by a function FBlocks.Get, its answer would also contain ranking. The MMI then can identify the function block with the highest ranking as a relevant partner.

In the example above, the MMI would get the function TA from tuner (slave) at ranking 0, and from the audio controller with ranking 1, and therefore would regard the audio controller as the relevant device.

2.2.10.2 Heredity of Functions

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

2.2.10.3 Deriving Devices/Device Hierarchy

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

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

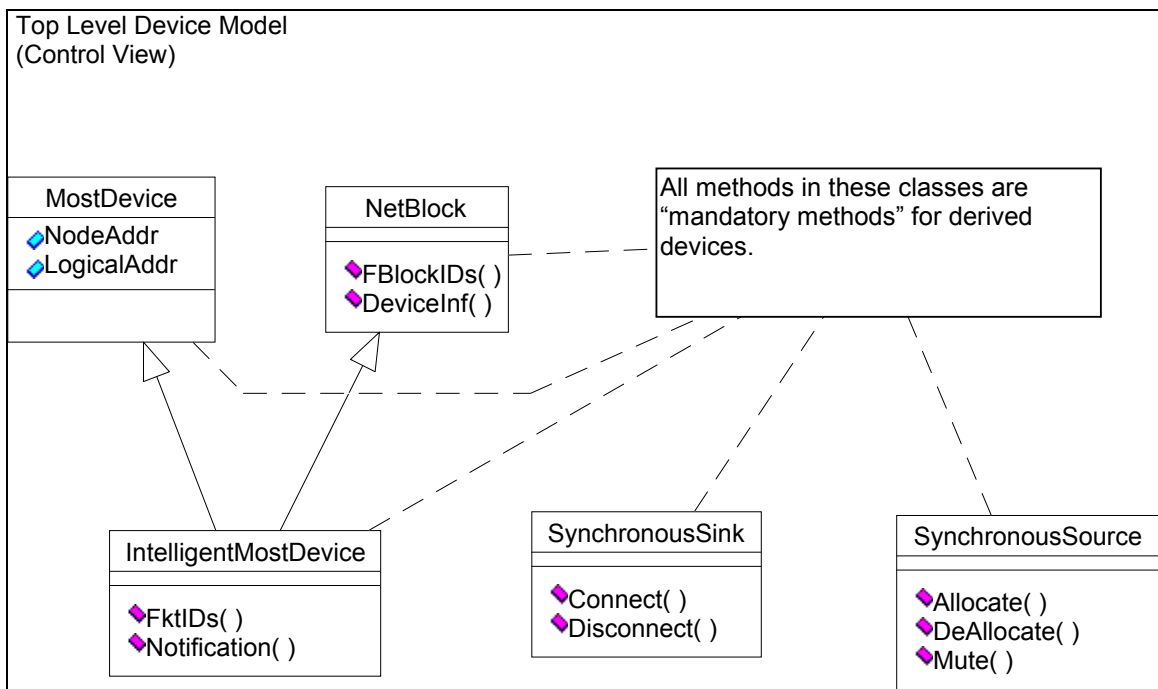


Figure 2-11: Highest layer of the device model

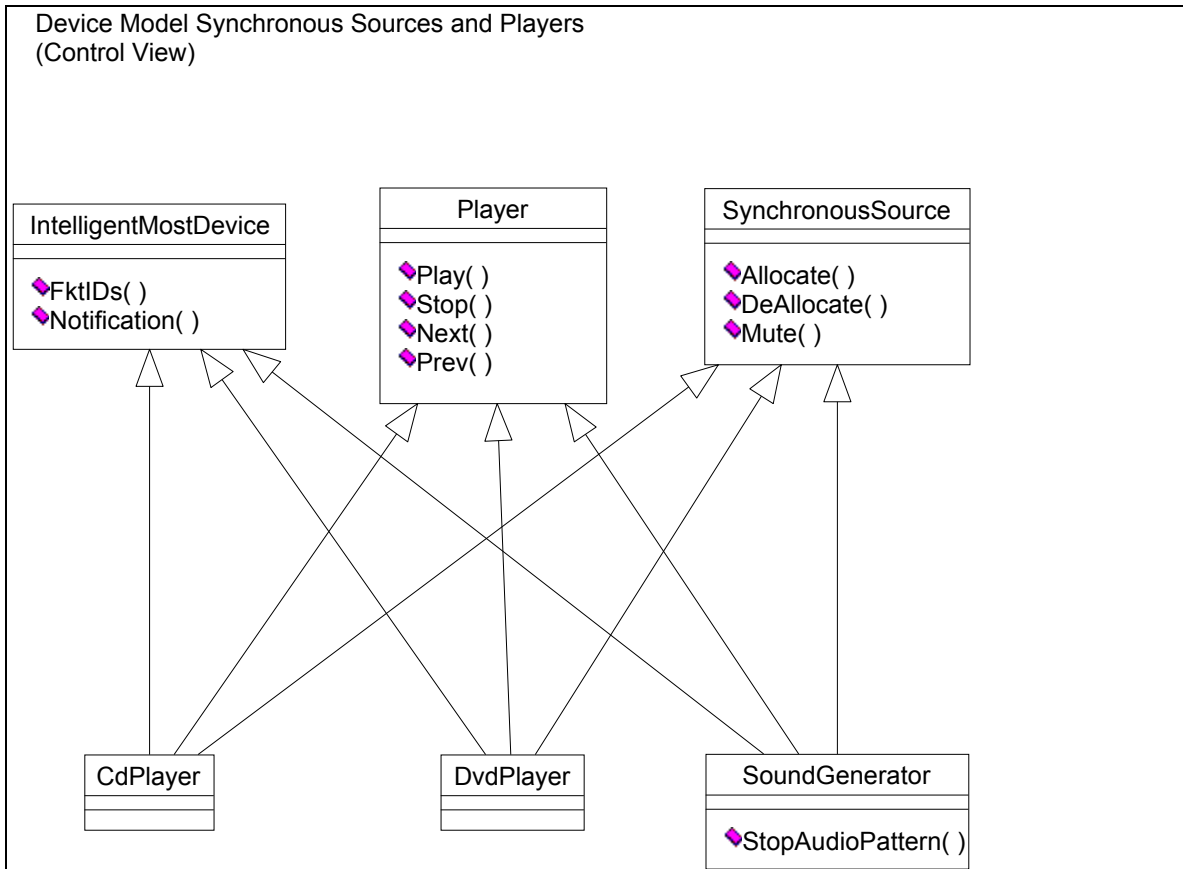


Figure 2-12: Device model for audio sources with player function

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

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

Device	Group of functions				
Master	MOST-Transceiver	+ NetBlock Master			
Slaves	%	+ NetBlock Slave			
all drives	%	%	+ Basic drive		
CDSingle	%	%	%	+ CD-Functions	
CDMulti	%	%	%	%	+ Changer

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

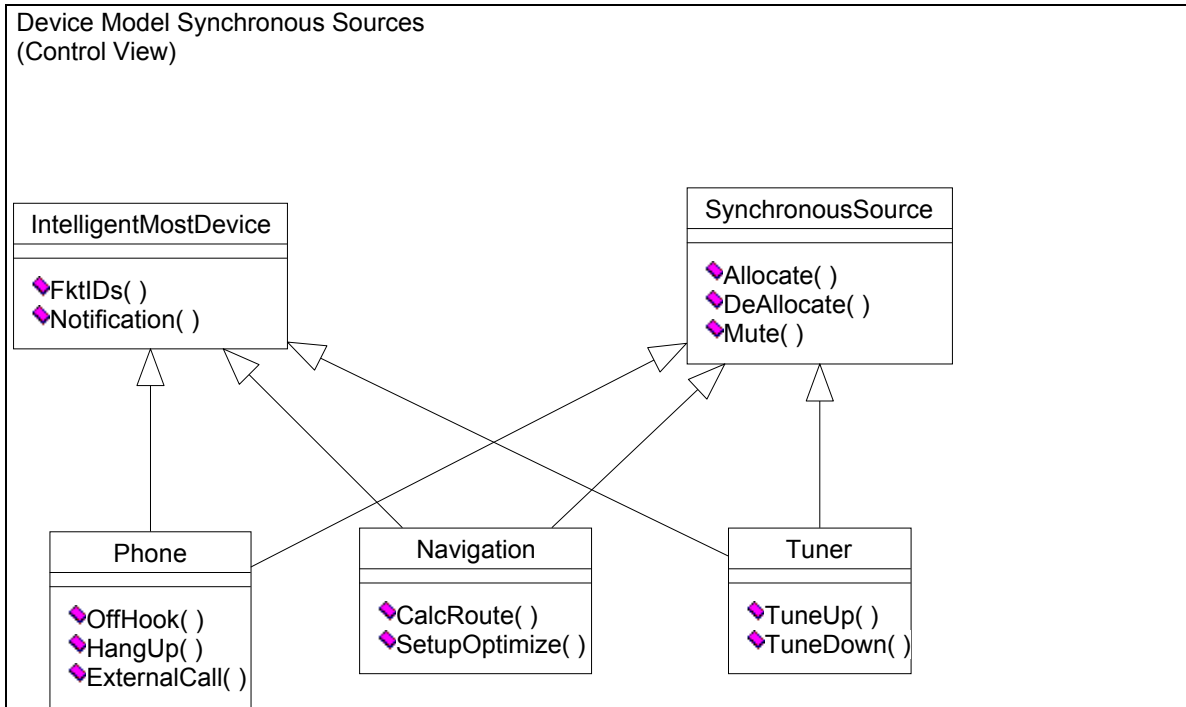


Figure 2-13: Device model for audio sources without player function

2.3 Protocols

2.3.1 Protocol Basics

As already described in section 2.2.1.2 on page 15, functions are addressed without considering the devices they belong to (on the application level). Functions are grouped together in function blocks with respect to their contents, making function blocks a good reference for localizing a certain function for an external application. A function is addressed in a function block. To distinguish between the different function blocks (FBlocks) and functions (Fkt) of a device, each function and function block has a name, or an identifier (ID) respectively:

FBlockID . FktID

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

FBlockID . FktID . OPType (Data)

2.3.2 Structure of MOST Protocols

The principal structure of protocols on the application layer is:

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

In addition to section 2.2.1.2 on page 15, three components were added: InstID, Length and DeviceID. The individual elements are explained below.

2.3.2.1 DeviceID

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

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

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

2.3.2.2 FBlockID

The FBlockID is the name of a special function block. Every function block with a special FBlockID must contain certain specific functions. In addition to those mandatory functions, it can contain other functions. The table below shows an (incomplete) collection of FBlockIDs:

Kind	FBlockID 8 Bit	Name	Explanation
Administration	0x0x		
	0x00	NetServices	No FBlock, stands for telegrams that are related to network tasks, and which therefore are not passed to application layer.
	0x01	NetBlock	
	0x02	NetworkMaster	
	0x03	Connection Master	
	0x04	Power Master	
	0x05	Vehicle	
	0x06	Diagnosis	
Operation	0x1x		
	0x10	Man Machine Interface	
	0x11	Speech Recognition	
Audio	0x2x		
	0x20	Audio Master	
	0x21	Audio DSP	
	0x22	Audio Amplifier	
Drives	0x3x		
	0x30	Audio Tape Recorder	
	0x31	Audio Disk Player	
	0x32	ROM Disk Player	
	0x33	Multimedia Disk Player	
	0x34	Versatile Disk Player	
Receiver	0x4x		
	0x40	AM/FM Tuner	
	0x41	DAB Tuner	
	0x42	TV Tuner	
Communication	0x5x		
	0x50	Telephone fix	
	0x51	Telephone mobile	
	0x52	Navigation System	
Video	0x6x		
	0x60	Display	
	0x61	Camera	
	0x62	Video Tape Recorder	
	0xC8	Reserved	
	0xF0..0xFE	Proprietary	
	0xFF	All	

Table 2-3: FBlockIDs.

2.3.2.3 InstID

It may happen that there are several equal function blocks (Instances) with the same FBlockID in one system (two CD changers, four active speakers, several diagnosis blocks, etc.). In order to address the function blocks unambiguously, the FBlockID is complemented by an instance ID (InstID), which differentiates equal function blocks in the system. The InstID consists of 8bits.

Predefinition:

0x00	Any Instance
0xFF	All function blocks having the same FBlockID.

At first (in an initialized system), every function block has instance ID 0x00. This value is chosen, since the FBlock either knows that it is unique, or since it cannot know about the existence of a "twin" in the system. In the latter case, the InstID must be assigned by a higher intelligence during a configuration run. The assignment can be encoded in a fixed way, or can be done, e.g., by a tester after having built up the system. The last case is more difficult.

The InstID therefore must be changeable. Every NetBlock has the property FBlockIDs which contains, among others things, the InstID. The following protocol will change an InstID from value OldInstID to value NewInstID:

```
Controller -> Slave : NetBlock.0.FBlockIDs.Set (FBlockID, OldInstID, NewInstID)
```

If each of the equal function blocks is located in a different device, they have the default InstID 0x00 (any instance). This can be changed easily, since the function blocks can be addressed by the logical node address, the FBlockID and the InstID in an unambiguous way. If there are several equal function blocks in one device, they cannot be addressed in an unambiguous way for changing InstID, since they have the same InstID. Therefore equal function blocks within one device will get an unambiguous InstID by default, that is, they will get numbered starting at 1. By using this default setting, every function block in the network is addressable by logical node address, FBlockID and InstID unambiguously at system start up, even if there are several equal FBlocks in one device.

When looking at the application level itself, where the logical node address will not be interpreted, the way of addressing as described above (FBlock.InstID) is not unambiguous. A higher intelligence must provide unambiguity in case of equal function blocks by setting unambiguous InstIDs on a system-wide level.

In principle, as long as the InstID provides the possibility to differentiate between equal function blocks, the InstID can be chosen in any way.

Please note:

The InstID distinguishes identical function blocks. 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).

2.3.2.4 FktID

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

The address range of FktIDs is subdivided in five sections:

1. **Coordination (0x000..0x1FF)**
Functions for administrative purposes in a function block.
2. **Mandatory (0x200..0x3FF)**
Functions that are mandatory for the application of the function block, like the basic drive in all function blocks describing drives.
3. **Extensions (0x400..0x9FF)**
Optional functions.
4. **Unique (0xA00..0xBFF)**
Functions that are defined unambiguously in the entire system.
Attention, these must be coordinated with the entire system!
5. **Proprietary (0xC00..0xFFE)**
Functions that are specific for manufacturers and which are confidential.

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

0x000	FktIDs	Reports the FktIDs of all functions contained in the FBlock (refer to section 2.3.9 on page 56).
0x001	Notification	Distribution list for events (refer to section 2.3.12 on page 81).

2.3.2.5 OPType

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

OPType	For properties	For methods
Commands:		
0	Set	Start
1	Get	Abort
2	SetGet	StartResult
3	Increment	--
4	Decrement	--
5	GetInterface	GetInterface
6	Reserved	StartAck
7	--	--
8	--	--
Reports:		
9	--	ErrorAck
A	--	ProcessingAck
B	--	Processing
C	Status	Result
D	--	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 the following table:

ErrorCode Data[0] On ErrorAck Data[2]	Description	ErrorInfo Data[1]..Data[n] On ErrorAck Data[3]..Data[n]	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. 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. Interpretation will be stopped then.
0x08	Parameter missing	--	No Info
0x09	Too many parameters	--	No Info
0x0A	Secondary Node	Return Address of Primary	Address of that node which is responsible for the secondary node sending the error
0x20	Function specific After this error code, any function specific ErrorInfo can be sent. Some, with general character, are suggested here.	0x01	Buffer overflow
		0x02	List overflow
		0x03	Element overflow
		0x04	Value not available
0x40	Busy Function is available, but is busy	--	No Info
0x41	Not available Function is implemented in principle, but is not available at the moment	--	No Info
0x42	Processing Error	--	No Info

Table 2-5: Error codes and additional information

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

1) Syntax error:

A syntax error occurs, if e.g. a function is accessed that does not exist, or if a not implemented OPType is called. Syntax error 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. The NetServices provide automatic syntax checking and reporting.

2) Application error – Parameter error:

The specified length does not match the actual length of the data field. There have been not enough, too many parameters, or one parameter is out of range. Parameter errors are reported by the ErrorCodes 0x05, 0x06, 0x08, and 0x09. Recognition of parameter errors is the task of application layer. The NetServices give support for reporting. Messages are only accepted when being completely correct. This means especially, that the length of the parameter area must be correct. The only exception is the handling of arrays that are too short (refer to section 2.3.11.2 on page 65).

3) Application error – Temporarily not available:

In some cases it may happen, that the message is correct, but the execution is not possible at the moment. The following distinction of cases must be performed:

- It may be that both methods and properties are implemented, but cannot be executed due to operation status. An example for a method would be SMSSend of the telephone, which cannot be executed if the bus is not available. In case of being called anyhow, it would report an OPType error at error code 0x41 “not available”. In such a case, the application can supervise the status of the telephone and may repeat the sending of the SMS as soon as the network is available again.
- A method can be available, but may be busy at the moment. So it would be possible, that method SMSSend of the telephone is busy in sending another SMS. In that case an error code 0x40 “busy” would be reported. Here, the application may perform retries. This case can only occur in connection with methods.
- A property represents a memory area, which is written by Set, or read by Get. According to definition this memory area cannot be “busy”. It is solely possible that a value is within the valid range, but is not selectable at the moment. An example can be property DeckStatus of the CD drive, which cannot be set to “Play” if there is no CD loaded. This would generate an error code 0x07 “parameter not available”.

4) Application error – General execution error:

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

5) Application error – Specific execution error:

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

The examination and processing of errors is done in the logical and temporary sequence as described above and in Figure 2-14 (below).

6) Application error – Error secondary node:

In a MOST Device it is possible to have two MOST Transceivers. On one of them (the primary node) the NetServices are running. On the other one (the secondary node), no NetServices are running. Nevertheless, all timing constraints which apply for “normal” MOST nodes (e.g. those with NetServices), must be fulfilled by secondary nodes too. The address of the secondary node should be determined and initialized too. In case a secondary node receives any control message, it replies with an Error “secondary node”. The reply contains the address of the primary node, which is responsible for that secondary node. In addition to that, the reply shall appear to be sent by NetBlock (with FBlockID=0x00, InstID=0x00) and from FktIDs function (FktID = 0x000).

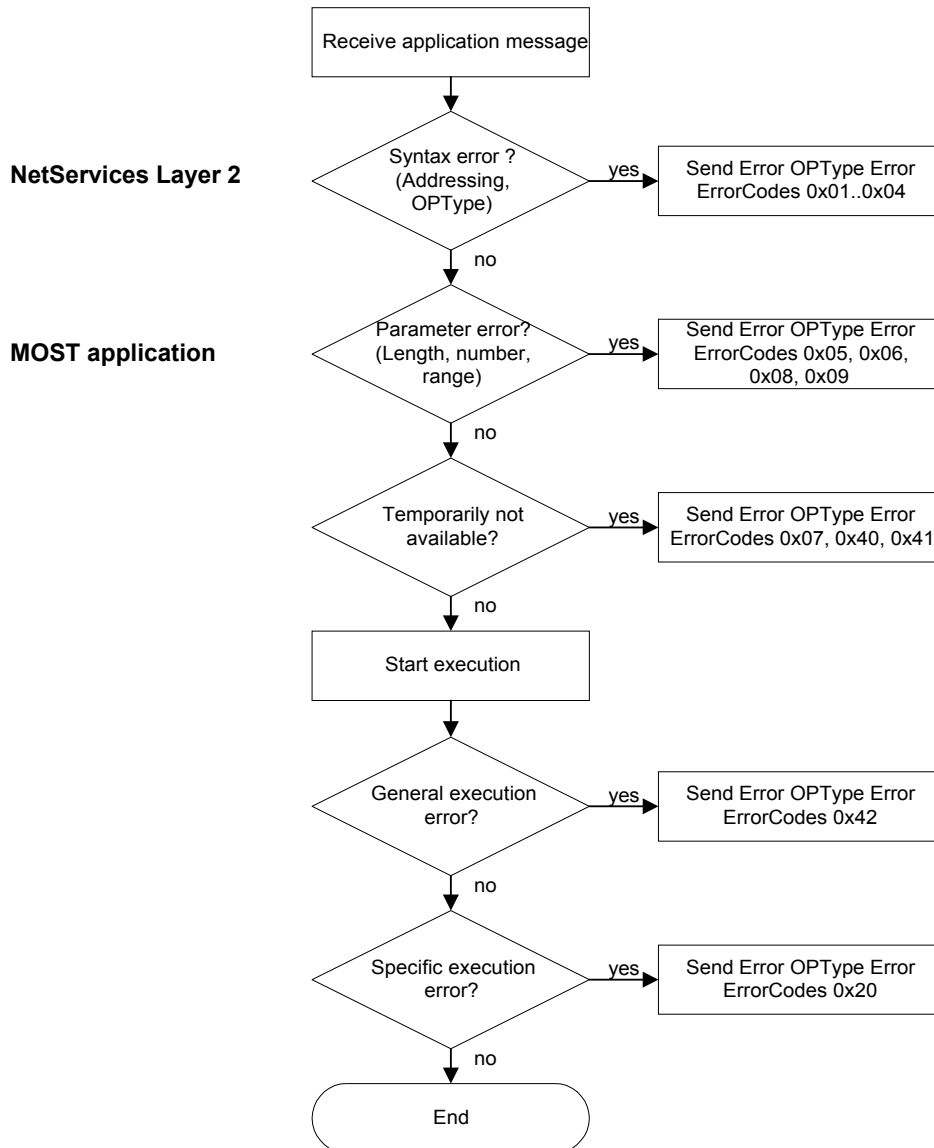


Figure 2-14: Processing of messages including error check on different layers.

Of course there can 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.

Please note:

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

2.3.2.5.2 Start, Result, Processing, Error

By using Start, a controller triggers a method. In opposite to StartResult, it “trusts” in the execution and does not expect a reply. So neither a reply using Processing, nor one using Result will be generated. This approach is useful only for Methods that do not return results.

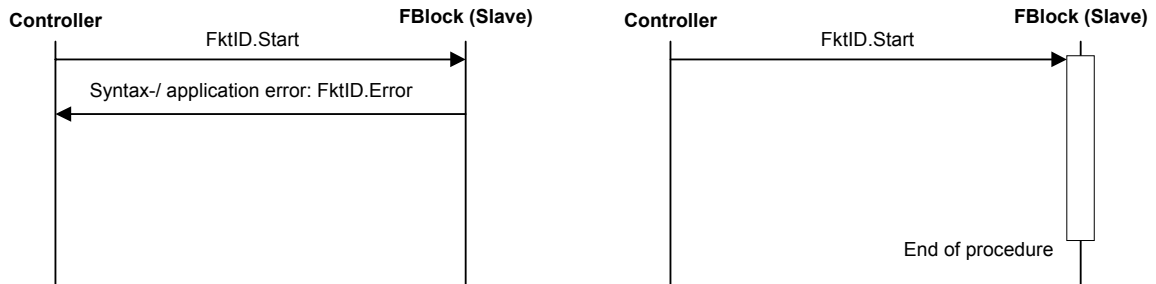


Figure 2-15: Sequences when using Start with and without error

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 100ms 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 can not generate a result 100ms after having received StartResult and if there is no application error, it replies after those 100ms by using “Processing”. After that the timer will be re-started and the procedure starts again. That means that in case of terminating the method within the 100ms, a reply “Result(<Parameter>)” will be sent. Otherwise “Processing” will be reported.

The controller evaluates the replies by using a timer interval of 200ms (compensation of eventual delays). In case that there is no reply within these 200 ms (Neither Result, nor Error, nor Processing), it assumes an error.

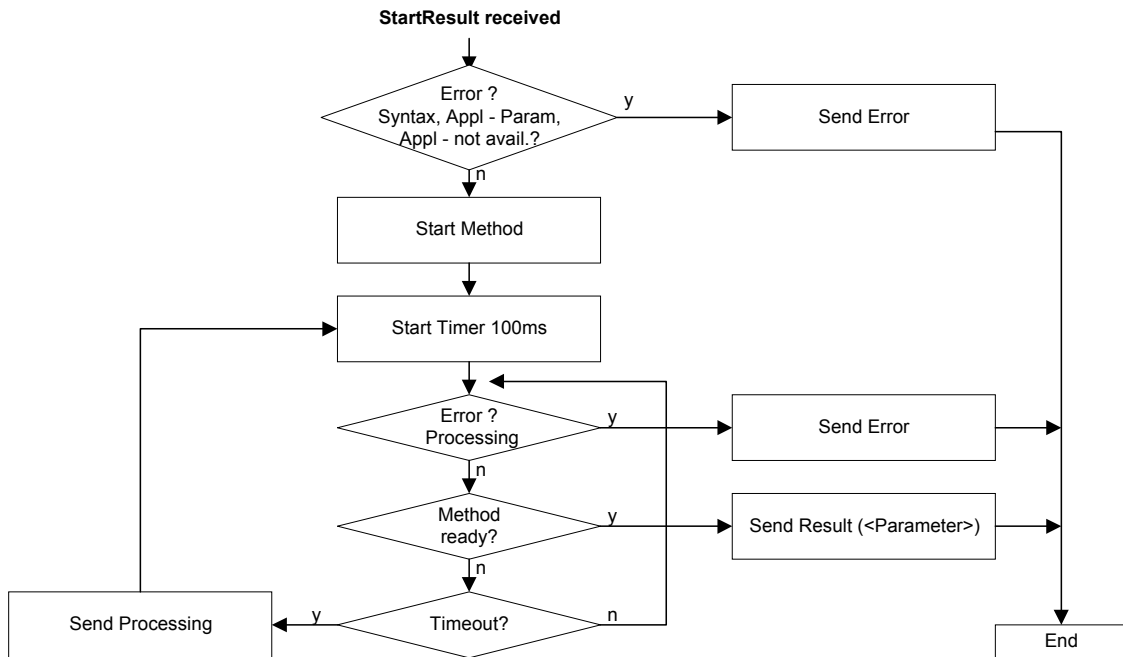


Figure 2-16: Flow for handling communication of methods (slave's side)

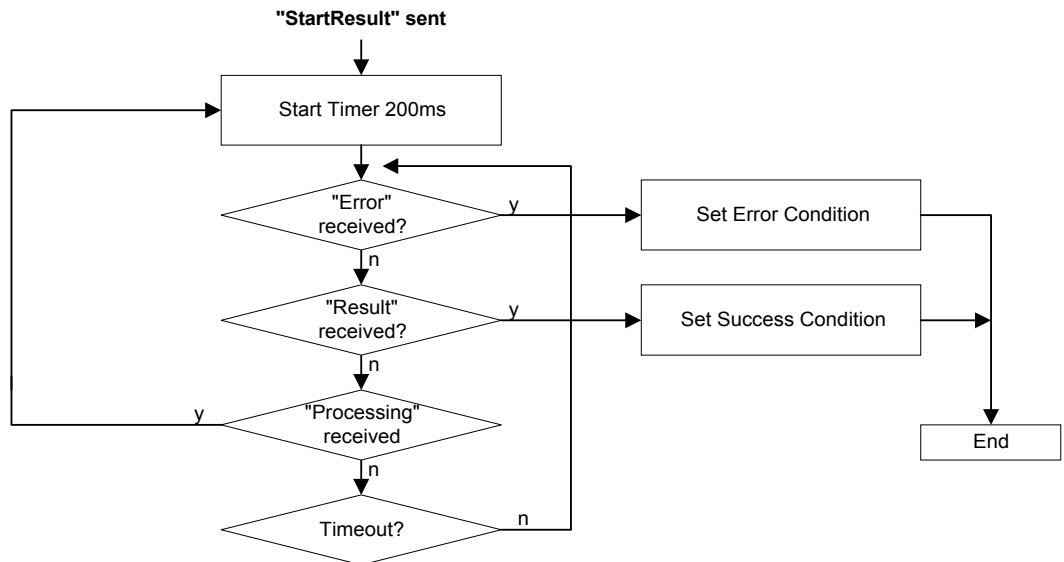


Figure 2-17: Flow for handling communication of methods (controller's side)

2.3.2.5.4 StartAck, ProcessingAck, ResultAck, and ErrorAck

The behavior is equal to that of StartResult, Processing, Result and Error (refer to section 2.3.2.5.3 on page 37). The single difference is, that the first parameter transports the SenderHandle (refer to section 2.3.6.2 on page 52).

2.3.2.5.5 Get, Status, Error

By using OPType Get, a controller asks for the status of a property. In case of a request by using Get, a reply using Status will be generated, if the syntax check has shown no errors. Otherwise Error will be returned. If the controller does not receive any reply 200ms 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 contents of a Property is changed. Set behaves equal to Start. This means that the controller "trusts" in the execution of its command and does not expect any reply (except eventual error reports). If the syntax check is ok, the command can be executed.

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

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

2.3.2.5.7 SetGet, Status, Error

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

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

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

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

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

This OPType is available for methods only. When used, Abort interrupts the execution of a method. It behaves like Start, which means that the Controller “trusts” in the execution and does not expect a reply.

2.3.2.6 Length

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

Length = 0x00 00	Data field of length 0
Length = 0x00 01	Data field of length 1 byte ...
Length = 0x0F FF	Data field of length 4095 Byte ...

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

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

In principle, the data field of a protocol on the application layer might have any length up to 4095 bytes. In a telegram on the control channel of the MOST bus, the maximum length is 12 bytes. So longer protocols must be segmented, i.e., be sent divided up in several telegrams. It should be kept in mind that even on the application level, the data fields of a protocol should exceed 12 bytes only in exceptional cases.

Within a data field, none, one, or multiple parameters in any combination of the following data types can be transported:

Bool-field	1 Byte	(If a variable of type Boolean is defined, a bit field of 8bits will be reserved, where the variable is represented by the LSB. The value 0b**** **0 means false and 0b**** **1 means true. It is possible to encode several variables of type Boolean in one bit field.
Enum	1 Byte	
unsigned Byte	1 Byte	
signed Byte	1 Byte	
unsigned Word	2 Byte	
signed Word	2 Byte	
unsigned Long	4 Byte	
signed Long	4 Byte	

They are transported MSB first. The sign is encoded in the most significant bit.

Since MOST provides enough bandwidth, 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:

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

Example 1:

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

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 1Hz)

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 10mm)

The exponent can be already known to the receiver of the parameter (controller), or it can be requested by the sender (function) of the value (refer to section 2.3.11 on page 58). It is not transported together with the parameter.

Since in many cases strings and uninterpreted data streams must be transported, the following additional basic data types are defined:

string	variable length; null terminated; The first byte is an identifier: 0x00 : UNICODE any other value : ASCII (e.g. in an Interface Description the maximum length of a string is specified. The length includes all characters, but it excludes byte 1 and the null for null termination)
stream	any data

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

2.3.3 Function Formats in Documentation

The protocol structure that was described above is valid for communication within devices, between the NetServices and a function block containing an application. 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 44). In most cases, only one instance of the function block is available in the system, and InstID can be omitted. Descriptions can also be simplified by omitting the Length.

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

Example:

Choosing track of the CD changer:

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

The tools of Oasis SiliconSystems (e.g., OptoLyzer4MOST[®] Professional, MOSTRapidControl) use a special form of syntax tree for automatic generation and disassembling of messages. In order to generate syntax trees automatically, these tools have a link to the catalog database.

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 MOSTNetwork, which is described in detail below.

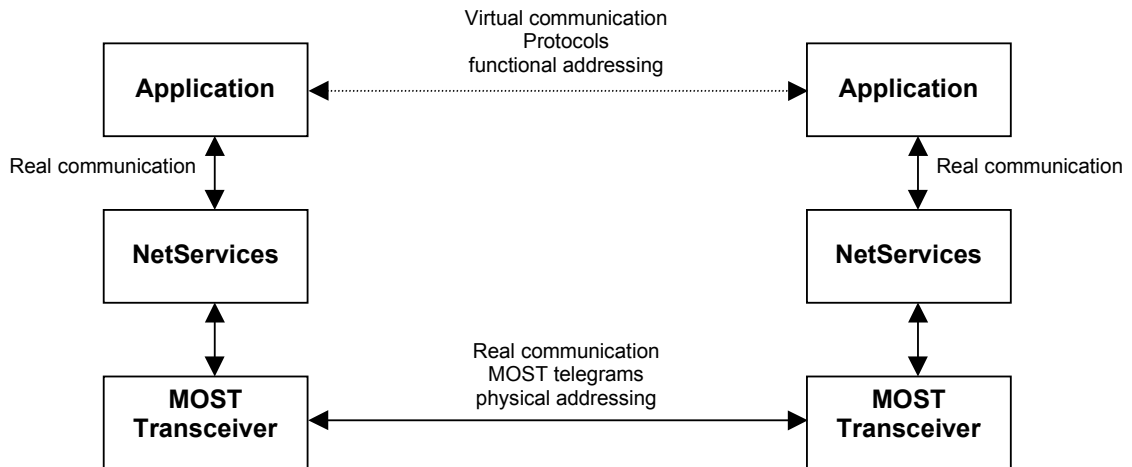


Figure 2-18: Virtual communication between two devices on application layer and real comm. via network

All application protocols are finally transferred via the control channel of the MOSTNetwork. From the application's point of view, all protocols are passed on to the NetServices. 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 MOSTNetwork, nodes, or devices, are addressed. In order to transport a protocol to a function block, the MOST telegrams are provided with the address of the device that contains the function block.

Here, the entire data flow of an interaction between two devices via the network layer is described. One device controls the functions of the other. The figure below shows the properties of a function block with the FBlockID CD and the InstID 0 (Any instance). The function block is found in device CD Player with the physical MOST address CDC.

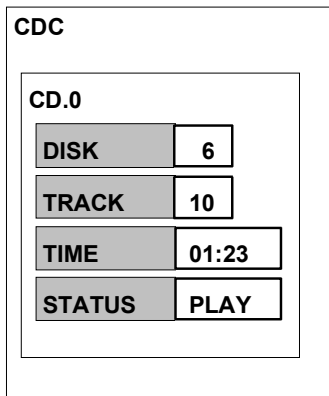


Figure 2-19: Device with MOST address CDC, a function block CD Player with FBlockID CD, and its functions.

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

`CD.0.Track.Set(10)`

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

`FFFF.CD.0.Track.Set(10)`

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

`CDC.CD.0.Track.Set(10)`

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

`MMI.CDC.CD.0.Track.Set(10)`

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

`MMI.CD.0.Track.Set(10)`

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

`MMI.CD.0.Track.Status(10)`

Based on this, the NetServices builds the following telegram:

`CDC.MMI.CD.0.Track.Status(10)`

In the MMI the receiver's address is removed and the protocol is passed to the application:

`CDC.CD.0.Track.Status (10)`

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

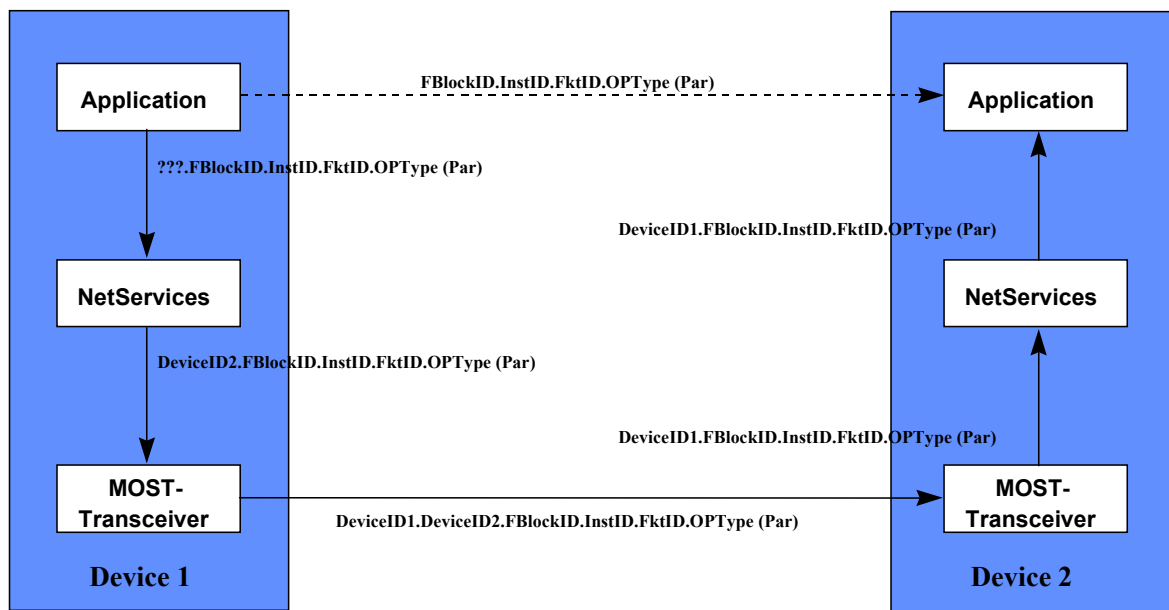


Figure 2-20: Communication between two devices via the different layers

2.3.6 Controller/Slave Communication

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

2.3.6.1 Communication With Properties Using Shadows

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

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

The properties of a device should describe the current operation status completely at any time. The figure below shows the properties of a function block CD changer with FBlockID CD and the unspecific InstID 0 in the device with the MOST address CDC.

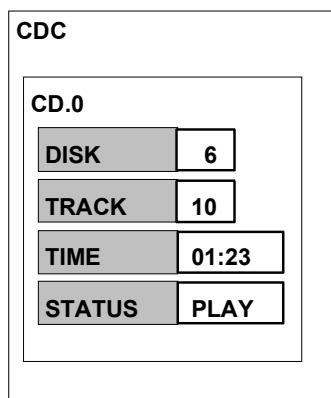


Figure 2-21: Example for a slave device

Operation status of the player is determined by the properties Disk (number of loaded CD), Track, Time, and Status (Play, Stop, Forward, Rewind, 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:

```
MMI->CDC: CD.0.Track.Set(10)
```

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

```
CDC->MMI: CD.0.Track.Status(10)
```

By sending this protocol, the player can be stopped:

```
MMI->CDC: CD.0.Status.Set(Stop)
```

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

```
CDC->MMI: CD.0.Status.Status(Stop)
```

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

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

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

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

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

```
CDC ->??? : CD.0.STATUS.STATUS(STOP)      Status of CD in CDC
CDC1->??? : CD.1.STATUS.STATUS(STOP)      Status of CD in CDC1
CDC2->??? : CD.2.STATUS.STATUS(STOP)      Status of CD in CDC2
CDC->??? : CD.1.STATUS.STATUS(STOP)      Status of 1st Player in CDC
CDC->??? : CD.2.STATUS.STATUS(STOP)      Status of 2nd Player in CDC
```

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

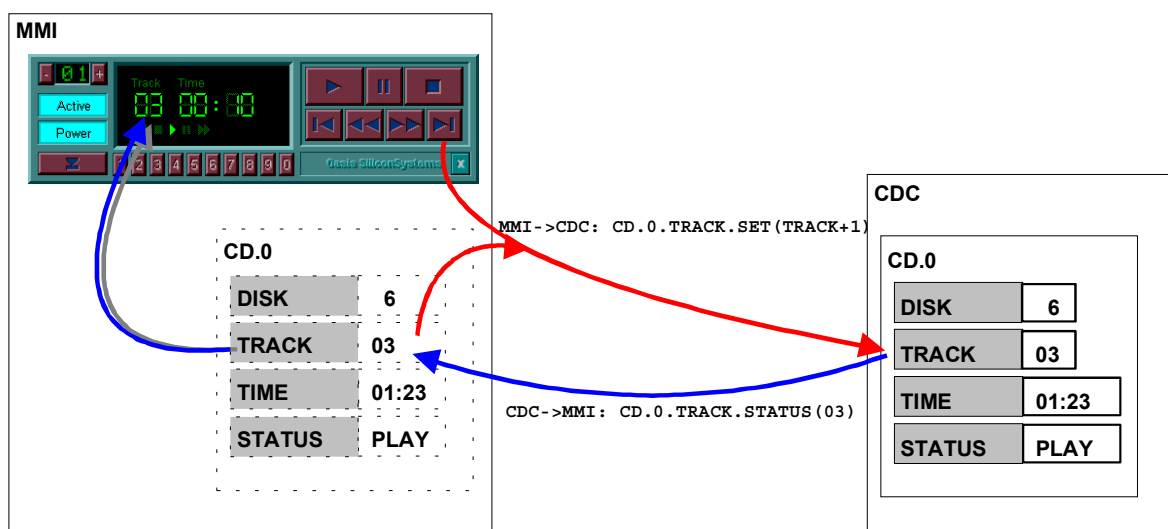


Figure 2-22: Virtual illustration of the controlled properties in the control device

The MMI shown in Figure 2-22 has an image of all properties of CDC (slave device) represented by the variables Disc, Track, Time and Status. These variables are required to store the display values, and can be used for control purposes too. The example shows the flow of communication when using the “Next track” button.

On a click onto the button, the MMI takes the contents of its local variable Track, increments it by one, and sends the protocol CD.0.Track.Set(Track+1) to device CDC. After the player has changed track, it replies by sending protocol CD.0.Track.Status(3). Addressing of the response is equal to the addressing of the command, except the address, since the answer is sent to a (virtual) identical function block. Variable Track reacts only on that protocol and stores the new value. The change of variable Track causes the MMI to update its display.

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

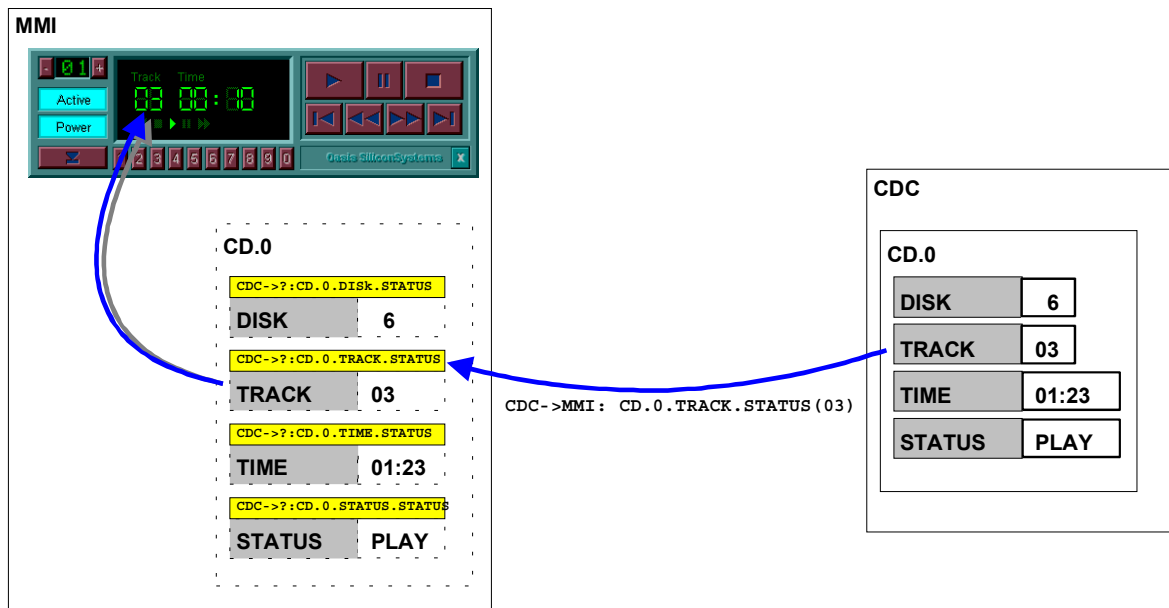


Figure 2-23: Unambiguous assignment between protocol and variable.

The figure below shows the advantage of this approach when controlling multiple devices. The MMI 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 MMI. 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.

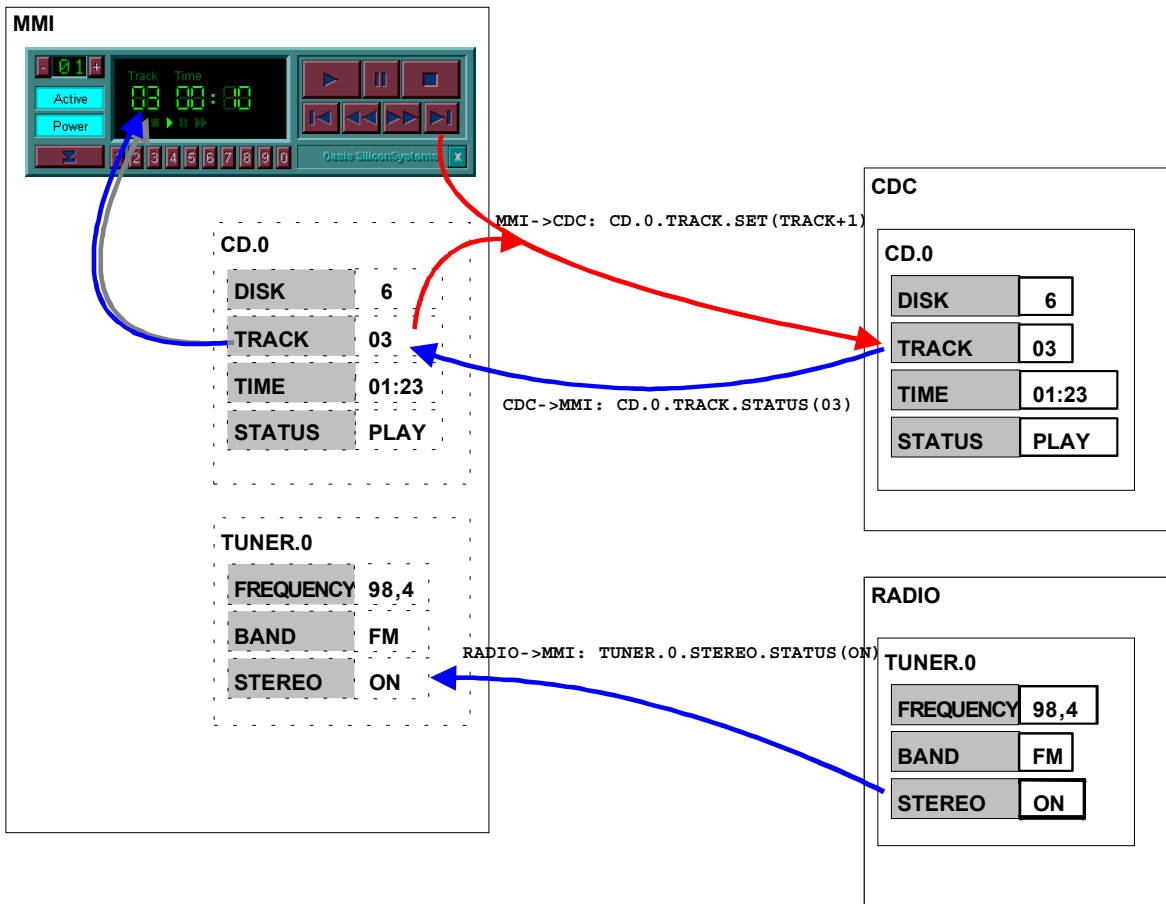


Figure 2-24: Controlling multiple devices.

A similar case could be imagined, if several identical CD players are available in the network. Operation mode of the MMI 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 an MMI would contain two sets of variables (shadows), one for each CD player. The variables for CD.1 react only upon protocols of CD.1, while the variables for CD.2 react only upon protocols of CD.2. If both of the function blocks are located in one device, handling would be identical.

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

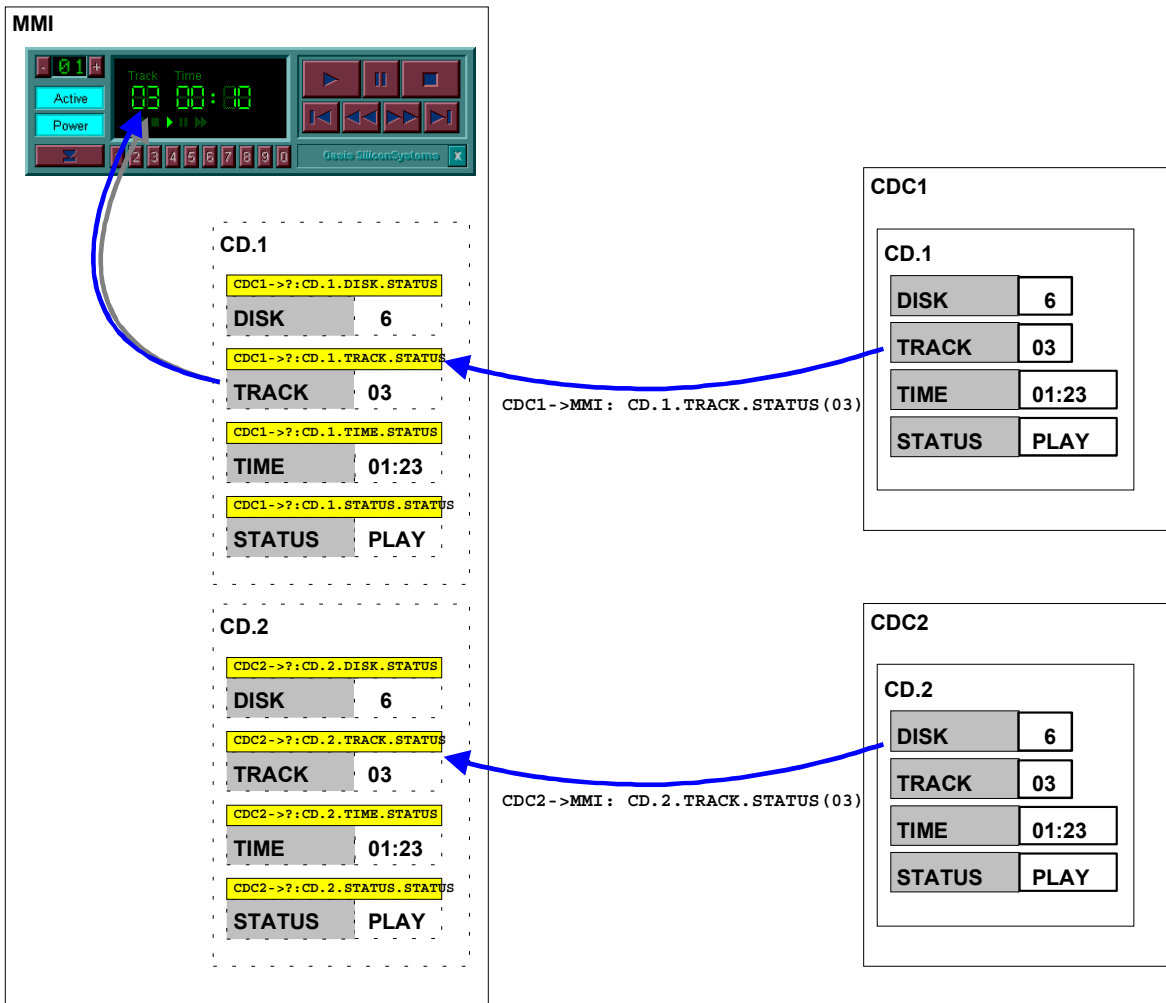


Figure 2-25: Controlling two identical devices.

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

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

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

On more complex control devices, this approach can be optimized by filtering the protocols step by step. The figure below shows an MMI 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.

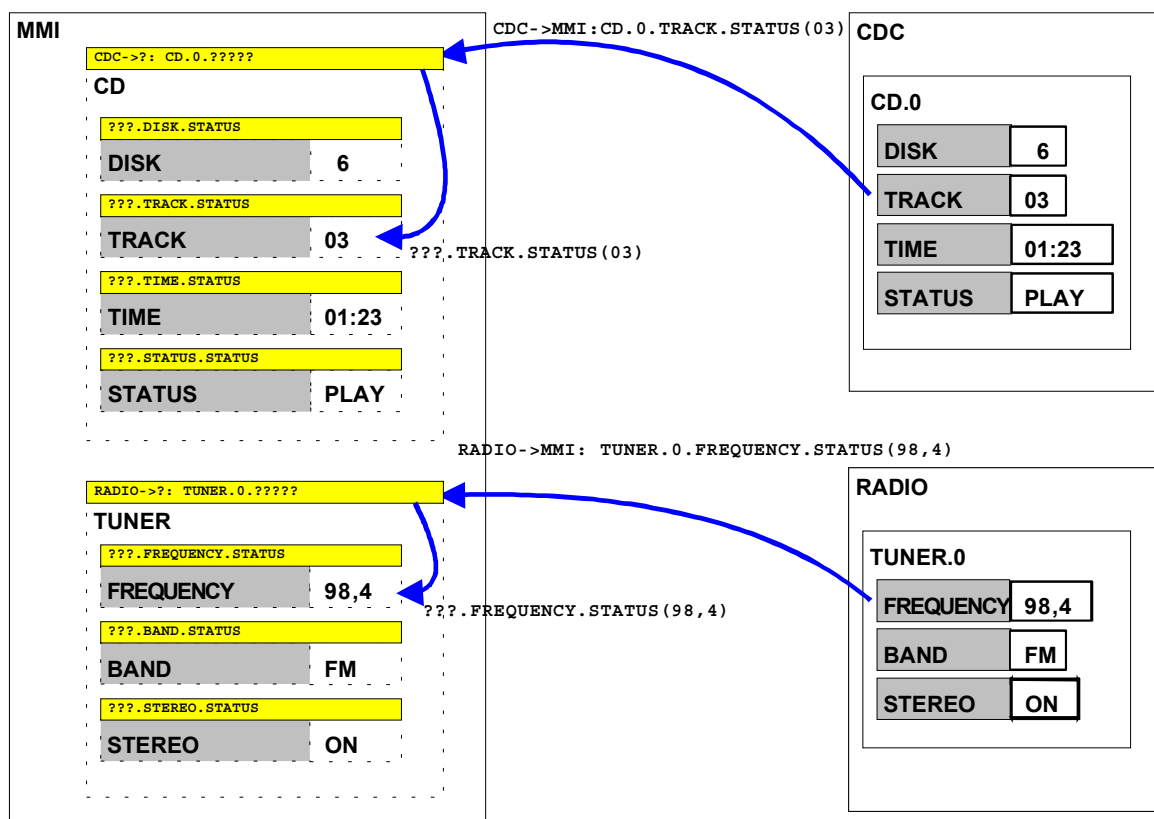


Figure 2-26: Hierarchical structure of the protocol filter (command interpreter).

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

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

2.3.6.2 Communication With Methods

2.3.6.2.1 Standard Case

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

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

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

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

2.3.6.2.2 Special Case Using Routing

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

In addition to that, in methods a process is triggered, which has a longer processing time. The Controller might 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 MMI, 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.

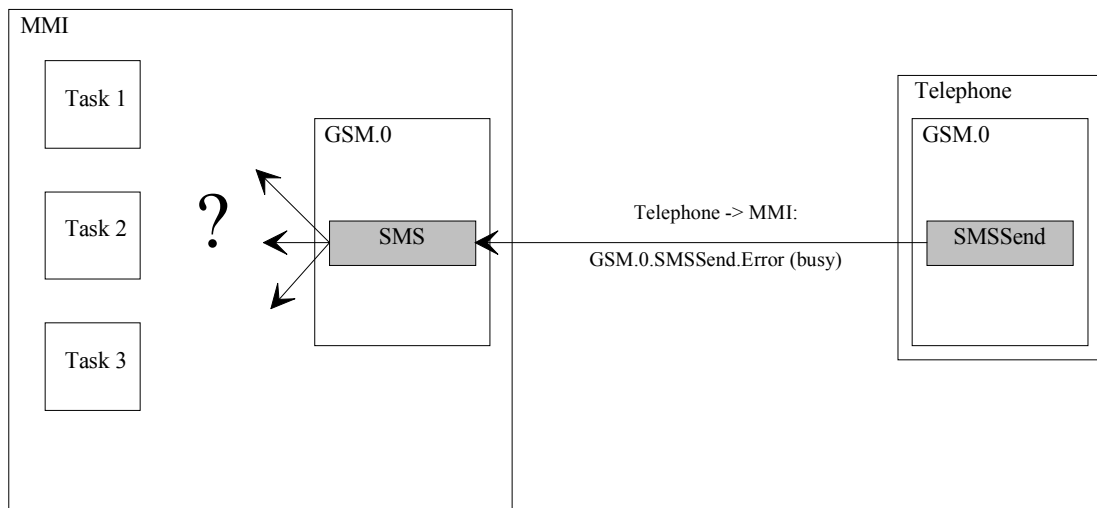


Figure 2-27: Routing answers in case of multiple tasks (in one controller) using one function.

To provide routing in such cases, the OPTypes StartAck, 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 StartAck 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 might look like:

```
Controller -> Slave: Telephone.0.SMSSend.StartAck (SenderHandle1.SMSData)
```

After successful transmission, Task 1 gets:

```
Slave -> Controller: Telephone.0.SMSSend.ResultAck (SenderHandle1)
```

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

```
Controller -> Slave: Telephone.0.SMSSend.StartAck (SenderHandle3.SMSData)
```

And it then gets in return:

```
Slave -> Controller: Telephone.0.SMSSend.ErrorAck (SenderHandle3.ErrorCode="Busy")
```

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

2.3.7 Seeking Communication Partner

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

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

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

The network master contains the central registry, which represents an image of the physical and logical system configuration. It answers with a list of all matching entries of the central registry with physical and functional address:

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

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

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

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

2.3.8 Requesting Function Block Information from a Device

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

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

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

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

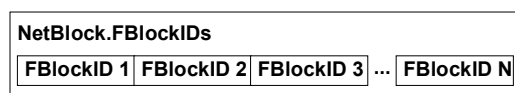


Figure 2-28: reading the function blocks of a device from NetBlock

2.3.9 Requesting Functions from a Function Block

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

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

Within a function block, FktIDs between 0x000 and 0xFFF (4096 different FktIDs) can be available. The FktIDs are assigned as described in 2.3.2 on page 29. This raises the problem of a compact response, if the functions contained in a function block are requested. It is solved by a mechanism derived from the run length encoding. A bit field is built where the first bit is set to 1 if FktID 0x000 is available, the second bit is set to 1 if FktID 0x001 is available, and so on. Such a bit field might 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.

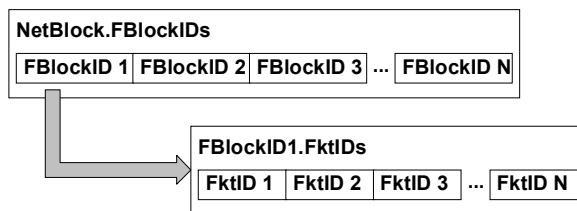


Figure 2-29: Requesting the functions contained in an application block.

2.3.10 Transmitting The Function Interface

2.3.10.1 Principle

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

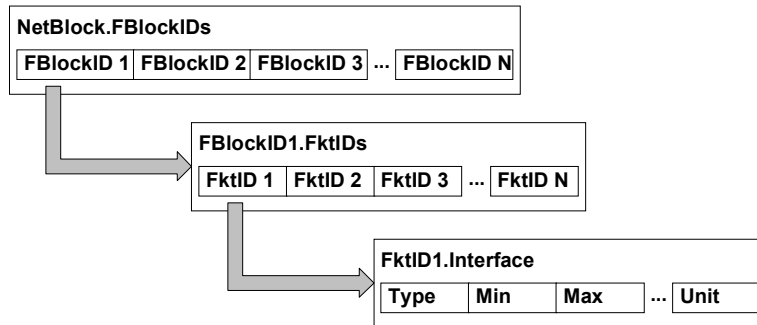


Figure 2-30: Requesting the function interface of a function

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

```

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

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

2.3.10.2 Realization Of The Ability To Extract The Function Interface

In the function catalog, every interface of classified functions is described. By doing this, a classified definition of application protocols, as well as a uniform description, is possible, which can be based onto a few classes.

Class:	8 Bit	0x00	Unclassified method
		0x10	Unclassified property
		0x11	Switch
		0x12	Number
		0x13	Text
		0x14	Enumeration
		0x15	Array refer to section 2.3.11.2.2 on page 68.
		0x16	Record refer to section 2.3.11.2.1 on page 66.
		0x17	Dynamic Array refer to section 2.3.11.2.3 on page 71.
		0x18	Long Array refer to section 2.3.11.2.4 on page 73.
		0xFF	Abort (No further specifications behind this location)
OPTypes:	16 Bit		Bit field of available (1 = OPType available)
Name:			Name of function as null terminated string.

2.3.11.1.1 Function Class Switch

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

Bool-field: 1 Byte 0 for off, 1 for on

Example: RDSOnOff in AM/FMTuner1

```

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

Upload interface:

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

Setting RDS = OFF:

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

2.3.11.1.2 Function Class Number

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

DataType: uns. Byte Type of variable:
 0x00 Unsigned Byte
 0x01 Signed Byte
 0x02 Unsigned Word
 0x03 Signed Word
 0x04 Unsigned Long
 0x05 Signed Long

Exponent: Signed Byte Position of decimal point; Value = Number * 10^{Exponent}

Min: Minimum value of variable of type *DataType*

Max: Maximum value of variable of type *DataType*

Step: Step width for adjusting type *DataType*

NSteps: uns. Byte Number of steps for adjusting

Units: uns. Byte Unit

Unit	Encoding
none	0x00
Distance:	
cm	0x01
m	0x02
km	0x03
mils	0x04
Time:	
us (Micro second)	0x10
ms (Millisecond)	0x11
s (Second)	0x12
min (Minute)	0x13
h (Hour)	0x14
d (day)	0x15
mon (Month)	0x16
a (Year)	0x17
Frequency:	
1/min	0x20
Hz	0x21
kHz	0x22
MHz	0x23
Volume:	
l (Liter)	0x30
gal (UK)	0x31
gal (US)	0x32

Unit	Encoding
Consumption:	
l/100km	0x40
mils/gal	0x41
km/l	0x42
Speed:	
km/h	0x50
mils/h	0x51
Temperature:	
°C	0x60
F	0x61
Volume:	
dB	0x70
Voltage:	
mV	0x80
V	0x81
Current:	
mA	0x90
A	0x91

Table 2-7: Available units

2.3.11.1.3 Function Class Text

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

MaxSize: uns. Byte maximum length of string

2.3.11.2 Properties with Multiple Variables

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

Functions with multiple variables can also be assigned to classes called array and record. In an array, variables 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.

2.3.11.2.1 Function Class Record

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

In the interface description of a record, the OPTypes are omitted, since they do not contain relevant information. OPTypes are relevant only in basic types.

NElements uns. Byte Number of elements in Record

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

Please note:

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

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

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

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

- x=y=0,
The operation is related to the entire record.
- x=(Position of array in record) AND y>0,
The operation is related to a "line" in the array.
- x=(Position of array in record) AND y=0,
The operation is related to the entire array.
- x<>(Position of array in record) AND y=0,
The operation is related to the respective element in the record.

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

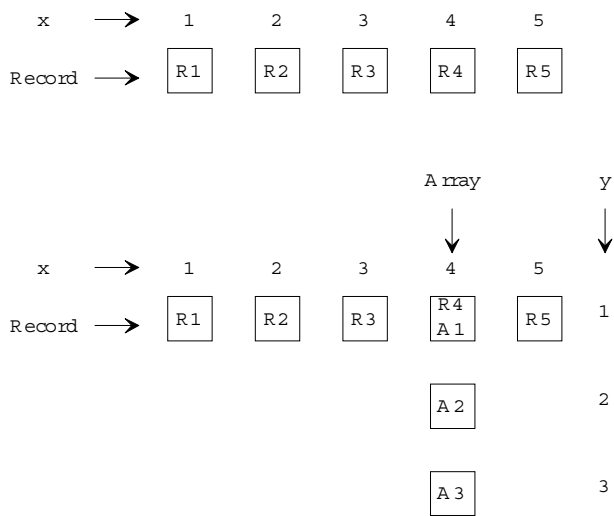


Figure 2-31: Meaning of position x in record (above) and of position y in a record with array (below).

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

2.3.11.2.2 Function Class Array

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

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

Class	IntDesc
Switch	Class, OPTypes, Name
Number	Class, OPTypes, Name, Units , DataType , Exponent , Min , Max , Step
Text	Class, OPTypes, Name, MaxSize
Enumeration	Class, OPTypes, Name, Size , Name1 , Name2 ,...
Array	Class, Name, NElements , IntDesc
Record	Class, Name, NElements , IntDesc1 , IntDesc2 ...

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

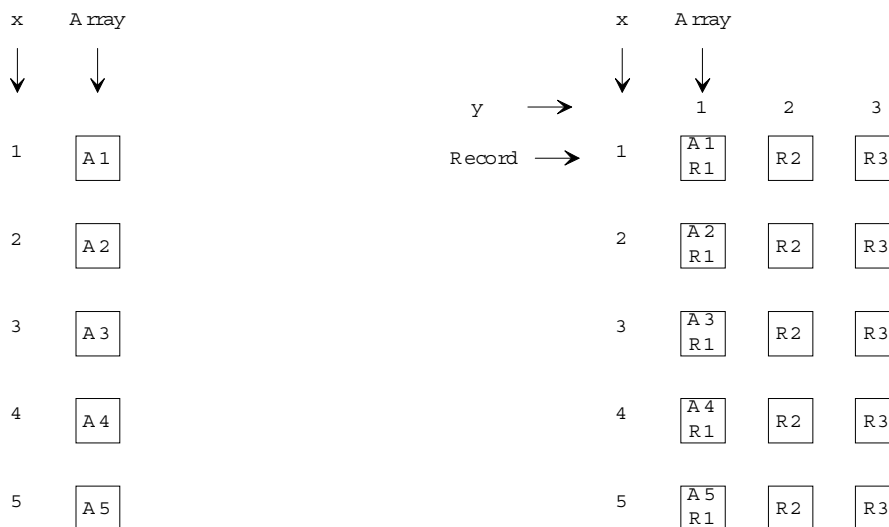


Figure 2-32: Position x in case of an array of basic type (left), and y in case of an array of record (right).

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 might update the signal qualities in a station list that was transferred earlier. It must take care to make sure that the signal quality values are assigned to the correct stations.

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

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

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

Examples:

Disk information in CD changer:

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

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

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

If a disk is not available, this can be recognized by TotalTime and NTracks containing 0x00. When requesting the FI, the formal answer is:

```
AudioDiskPlayer.0.Magazine.Interface
```

(Flags. Class. Name. NMax.	Array of
Class. Name. NElements.	Record of
Class. OTypes. Name. MaxSize	Text
Class. OTypes. Name. Units, DataType, Exponent, Min, Max, Step	Number
Class. OTypes. 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. OTypes. "DiskTitle", FF
Number. OTypes. "TotalTime". Seconds. Word. 00. 00 00. FF FF. 00 01
Number. OTypes. "Tracks". 00. Unsigned Byte. 00. 01. 63. 01)
```

On request

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

one receives the title of the third disk. On request

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

the titles of all disks are returned.

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

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

On requesting the FI, the formal answer is:

```
AudioDiskPlayer.0.Disk.Interface
```

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

or more related to the contents:

```
AudioDiskPlayer.0.Magazine.Interface
```

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

Selecting In Arrays:

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

- n of N:
The selection here should be done by an individual parameter Selected of type Switch, which is used as prefix (Array of record of {Selected, ...}). The change in the status of the switch can be modified by Controller or Slave either single (Selected of a single line), or for an entire column (Selected of all lines). In principle this kind of selection can be used in case of 1 of N as well.
- 1 of N:
In case of 1 of N there is an alternative modeling which is less expensive with respect to communication than n of N. Here a property Selected is modeled, which points onto the selected line. The kind of pointer differs individually. So e.g. in case of station lists the pointer may point onto the PI of the station currently active. In other cases, the position may be more effective. This way can be very effective, if a single line shall be selected in several Arrays (e.g. an entry in all telephone directories).

2.3.11.2.3 Function Class Dynamic Array

The arrays described above are optimized with respect to a high data volume. Navigation is based on the fixed sequence of elements in the array (Position = PosX, PosY). The position will not be contained in the data field. In Dynamic Arrays this is not possible, since here, lines can be inserted or removed (the sequence may vary). So a special function class DynamicArray is introduced, where PosX will be replaced by a uniquely defined handle, the Tag of data type uns. Word. It is defined as first parameter in the record:

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

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

OPType	Parameter
Set	Tag, PosY, Data
Get	Tag, PosY
Status	Tag, PosY, Data
SetGet	Tag, PosY, Data
Increment	Tag, PosY, NSteps
Decrement	Tag, PosY, NSteps
GetInterface	
Interface	Refer to section 2.3.11.2.2 on page 68
Error	ErrorCode, ErrorInfo

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

Please note:

The Tag belongs to the data field. This means that it is returned at the start of every line. PosY = 0x01 denotes the Tag. With respect to consistency, accesses to a column are not reasonable.

Examples for positioning:

1. Array of Record of {Tag, EI1, EI2, EI3, EI4}
2. Tag = 0x00 00 and PosY = 0x00
3. Tag = 0x20 06 and PosY = 0x00
4. Tag = 0x6389 and PosY = 3

(1)					(2)					(3)					(4)				
Tag	EI1	EI2	EI3		Tag	EI1	EI2	EI3		Tag	EI1	EI2	EI3		Tag	EI1	EI2	EI3	
0356					0356					0356					0356				
3467					3467					3467					3467				
3624					3624					3624					3624				
2006					2006					2006					2006				
0101					0101					0101					0101				
6389					6389					6389					6389				
0900					0900					0900					0900				
3581					3581					3581					3581				
9023					9023					9023					9023				

Editing In DynamicArrays:

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

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

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

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

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

Examples:

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

2.3.11.2.4 Function Class LongArray

A Slave transfers the arrays and DynamicArrays (as described above) to the registered controller using shadows. In case of changes, the shadows then will be updated. In case of big arrays that are changed very often, this may not be practicable any longer (Amount of memory in Controller, transmission time, bus load). Here another model – LongArray – must be applied. Where to place the boundary between LongArray and DynamicArray, is a matter of an individual decision.

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

2.3.11.2.4.1 MotherArray

The MotherArray is structured like a function of class DynamicArray. So communication of DynamicArray is identical to the communication of MotherArray, but there are only the OPTypes GetInterface, Interface and Error available (refer to section 2.3.11.2.2 on page 68).

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

Tag	EI 1	EI 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

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 function catalog). In other function classes, where functions are instantiated in a static way, those functions describe fixed properties and methods of the Slave. Their state is identical for all controllers. With respect to its status, an ArrayWindow is strongly bound to a Controller. So there must be an individual ArrayWindow for each Controller. So it is possible that several MMIs have individual ArrayWindows to an address directory (MotherArray), which have different size and position.

So functions of class ArrayWindow are instantiated dynamically at runtime. Therefore a function block (that has a MotherArray, which shall be accessed by ArrayWindows), must provide a method CreateArrayWindow for instantiation, and a method DestroyArrayWindow (both of class Unclassified Method). The FkIID is used as instance handle, which is transferred from Slave to Controller during instantiation. The FkIIDs used here must be occupied in Layer II of the NetServices, since a dynamical extension of the command interpreter is not available.

Function	OPTypes	Parameter
CreateArrayWindow	StartAck	SenderHandle, FkIIDMotherArray, WindowSize
	ResultAck	SenderHandle, FkIIDArrayWindow
	ErrorAck	SenderHandle, ErrorCode, ErrorInfo
DestroyArrayWindow	StartAck	SenderHandle, FkIIDArrayWindow
	ResultAck	SenderHandle
	ErrorAck	SenderHandle, ErrorCode, ErrorInfo

FkIIDMotherArray FkIID of the MotherArray. It is not dynamic, since MotherArray is a property of class DynamicArray of the Slave

FkIIDArrayWindow FkIID of the ArrayWindow. It is generated dynamically and represents the object handle which is transferred during instantiation. A range for such dynamically generated FkIIDs is occupied in advance.

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

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

The status of an ArrayWindow is kept up to date in the Controller by using a shadow. Also in that case, it is the Slave's task to keep the shadow up to date. Here the notification mechanism of the NetServices cannot be used, since it is static. Notification for ArrayWindows is to be implemented at application level. For each ArrayWindow there is only one single shadow, which is located in the Controller that has instantiated it. The DeviceID of the Controller is transferred to the Slave during instantiation, so there is no need to implement a special notification mechanism for registering the controller.

By using the ArrayWindow, editing the MotherArray can be done in the conventional way:

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

Tag	EI 1	EI 2
6243	a	01
2100	b	02
5428	c	03
0101	d	04
3245	e	05
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

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

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

MotherArray

ArrayWindow

SetGet (9473, 03, B3)

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

Function	OPTypes	Parameter
ArrayWindow	Set	Refer to function class DynamicArray (section 2.3.11.2.3 on page 71)
	Get	Refer to function class DynamicArray (section 2.3.11.2.3 on page 71)
	Status	Refer to function class DynamicArray (section 2.3.11.2.3 on page 71)
	Increment	Refer to function class DynamicArray (section 2.3.11.2.3 on page 71)
	Decrement	Refer to function class DynamicArray (section 2.3.11.2.3 on page 71)
	Error	Refer to function class DynamicArray (section 2.3.11.2.3 on page 71)

2.3.11.2.4.3 Positioning An ArrayWindow On A MotherArray

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

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

FktID FktID of the ArrayWindow to be moved

Mode uns. Byte 00 Top
 01 Bottom
 02 Up
 03 Down
 04 Absolute

Top and Bottom:

Top and Bottom move the ArrayWindow to the start, or the end of the MotherArray respectively. The parameters Number and Tag are transferred as well, but they are unimportant.

Tag	EI 1	EI 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

MotherArray

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

ArrayWindow

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

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

5674	v	1C
9643	w	1D
6354	x	1E
3425	y	1F
1045	z	20

MoveAW.Start (FktIDBottom, 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 has "hit" the start or the end of the MotherArray, no error will be reported, if Up or Down try to move it out of the valid range.

Tag	EI 1	EI 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

MotherArray

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

ArrayWindow

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

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

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

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

Absolute:

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

Tag	EI 1	EI 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

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

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

MotherArray

ArrayWindow

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

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

`SearchAW.Start (FktID, PosY, Searchstring)`

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

Tag	EI 1	EI 2
6243	a	01
2100	b	02
5428	c	03
0101	d	04
3245	e	05
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

MotherArray

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

ArrayWindow

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

SearchAW.Start (FktID, 02, „c“)

2.3.11.3 Function Class For Methods

For methods there is only one function class, since methods may differ significantly with respect to the parameters transferred during Start and Result (in opposite to properties). Methods that have to transfer parameters in case of Start and/ or Result, belong to class "unclassified method" (0x00). They must be defined in a specific way.

The only function class for methods is function class Trigger. There are no parameters in case of Start/ StartResult, and it does not return parameters in case of Result or Processing.

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

2.3.12 Handling Message Notification

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

Please note:

Only properties can be admitted to the notification matrix!

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

Table 2-8: Notification matrix (x=notification activated)

The size of a notification matrix depends on the function block, on the number of properties, and on the number of device entries, each of which must be registered individually. The minimum number of device entries is three.

When taking into consideration that a DeviceID has 16bits, a FktID has 12bits, and that in some function blocks possibly all 64 possible nodes of the network must be registered, the notification matrix might 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 might be done in very economical ways, e.g., by pointers in every function object that point to DeviceIDs.
- In most cases it is sufficient if the notification matrix has only a few entries.
- Group addresses are allowed as DeviceID in the notification matrix.

For very simple function blocks, for example, a CD changer, it is sufficient if the notification matrix provides only three entries for DeviceIDs. A very efficient implementation is possible. For example, by using a group address, all MMIs 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 44, but in order to enter group addresses, the DeviceID is transmitted in the parameter field as well. Parameter Control (8 bits, only 4bits used) 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 2-9: 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 2-10: Protocols with different controls for making entries in the notification matrix, and the resulting entries.

Immediately after registration in the notification matrix, the controller receives the status reports of all functions it has activated as events. If a double registering occurs, that is, a device registers that has already been registered, the reports are sent as if the device has been registered for the first time. This also applies to registering with group addresses.

Deleting entries is done in a similar way.

If a controller desires to read information from the notification matrix, it sends:

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

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

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

Please note:

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

Error handling:

If no more registering is possible, function Notification answers:

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

ErrorCode is 0x20 (Function specific) then, and ErrorInfo is 0x01 (Buffer overflow).

Please note:

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

3 Network Section

3.1 MOSTTransceiver and its Internal Services

The MOSTTransceiver OS8104 of Oasis SiliconSystems provides extensive tools for operating the MOST bus simply and safely, and for the transmission of data of different origins. Based on these tools, higher layers are defined. For a more detailed description of the MOSTTransceiver please refer to [7]. The following sections give an overview of the features of the OS8104 that are available for simplifying the definition of higher layers.

3.1.1 Electrical Bypass (All Bypass)

If the bypass is closed, all signals received at the RX pin of the MOSTTransceiver are electrically connected to the RX pin (shortcut). In this state, the respective device is “invisible” to the network. The device will be considered for the automatic counting of bus components only after opening the bypass, which gives access to the bus.

After the MOSTTransceiver is reset, the bypass is closed. This allows a very fast startup of the system, especially on usage of an optical wakeup mechanism. The bypass must be opened by the controlling micro controller after wakeup of the component.

3.1.2 Source Data Bypass

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

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

3.1.3 Master/Slave, Active and Passive Components

Basically, a MOST system consists of up to 64 nodes with identical MOSTTransceivers. By configuration, any of the transceivers can be the timing master, all the others are slaves. The timing master provides generation and transporting of system clock, the frames, and blocks. All slave devices derive their clock from the MOST bus.

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

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

3.1.4 Data Transport

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

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

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

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

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

3.1.4.1 Blocks

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

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

3.1.4.2 Frames

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

For synchronization, two different bus node types are required. A timing master that generates the frames, and slave devices that synchronize to the master clock on the bus.

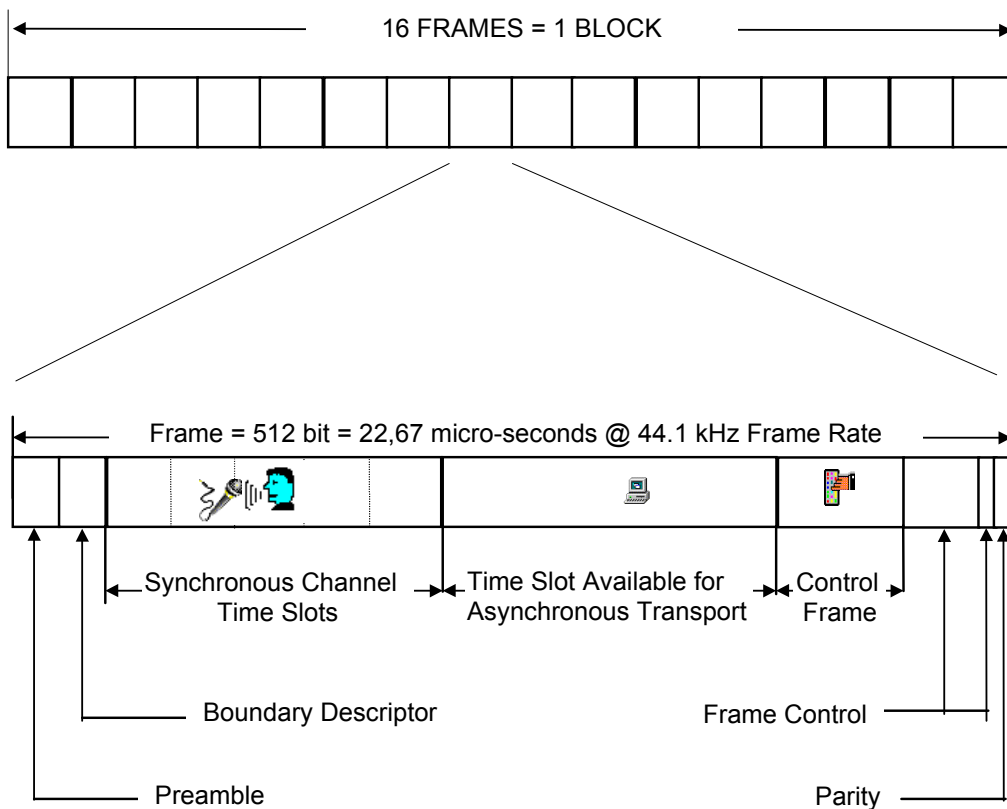


Figure 3-1: Structure of blocks and frames on the MOST bus.

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

Byte	Bit	Task
0	0-3	preamble
0	4-7	boundary descriptor (synchronous area count value)
1	8-15	data byte 0
2	16-23	data byte 1
⋮	⋮	⋮
60	480-487	data byte 59
61	488-495	control frame byte 0
62	496-503	control frame byte 1
63	504-510	frame control and status bits
63	511	parity bit

Table 3-1: Structure of the MOST frame

3.1.4.2.1 Preamble

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

For synchronization to a frame, two different mechanisms are used for slave and master nodes. For a slave node, the first reception of valid preambles after reset, power-up, or loss of lock indicates that phase lock on the input bit stream has been accomplished.

This method ensures that the slave node is phase- and frequency-locked to the bit stream, and hence the master node. In a master node, the transmitted bit stream is synchronized to an external timing source such as a crystal oscillator, SCK, FSU, or S/PDIF source.

Once all the nodes in the network have locked to the master's transmitted bit stream, the received bit stream has the correct frequency, but will be phase shifted with respect to the transmitted bit stream. This phase shift is due to delays from each active node, and additional accumulated delays due to tolerances in the phase lock within the slave nodes. The master node re-synchronizes the received data by the use of a PLL to lock onto the incoming bit stream, thereby re-synchronizing the incoming data to the proper bit alignment.

3.1.4.2.2 Boundary Descriptor

The boundary descriptor provides a flexible way of changing the bandwidth for synchronous and asynchronous data transmission. It represents the number of 4 byte blocks (quadlets) of data used for synchronous data. This value is used to determine the boundary between the synchronous and asynchronous data areas in the frame. A count value of 0 indicates no synchronous data and 15 quadlets of asynchronous data, while a count value of 15 indicates 15 quadlets of synchronous data and no asynchronous data.

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

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

The boundary descriptor is managed by the timing master of a MOSTNetwork. Please note that all synchronous connections must be re-built after having changed bSBC.

3.1.4.2.3 MOST System Control Bits

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

3.1.4.3 Source Data

3.1.4.3.1 Definition of Control Data and Source Data

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

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

3.1.4.3.2 Differentiating Synchronous and Asynchronous Data

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

3.1.4.3.3 Source Data Interface

The MOST_{Transceiver} 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 transceiver. For more detailed information please refer to [7].

3.1.4.3.4 Transparent Channels

In addition to the different transmission procedures, the MOST_{Network} provides a transparent interface (transparent port). This port is oversampled (depending on the system's sample rate, and on the sampling rate chosen for the transparent port) and routed via the network. Therefore source data port 1 (SR1 and SX1) is available. It provides, for example, the transparent transmission of a RS232 interface, i.e., without synchronizing RS232 to the bus.

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

3.1.4.3.5 Synchronous Area

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

Accessing this data is provided by time division multiplexing (TDM) and allocation of quasi-static physical channels for a certain period of time (e.g., while playing an audio source). The bandwidth for such a channel can be adjusted by allocating any number of bytes to one logical channel. The maximum number of bytes available in a synchronous channel is 60 bytes/frame, which is corresponding to 60 x 8 bits or 15 stereo channels of CD-quality audio. The typical frame rate is 44,100 frames/second.

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

3.1.4.3.6 Asynchronous (Packet Data) Area

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

Access to this type of data is provided in a token ring manner. Each node has fair access to this channel and its bandwidth can be controlled using the boundary descriptor in a step of four bytes (quadlets). The maximum packet length on an asynchronous channel when using the 48 bytes data link layer, is 48 bytes. In case of using an alternative data link layer, the maximum packet length is 1014. The data on this channel is CRC protected. The asynchronous message is defined as follows:

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

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

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

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

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

3.1.4.4 Control Data

3.1.4.4.1 Control Data Interface

The transmission of control data to and from the MOST_{Transceiver} is done via the control bus (at the control port). Depending on the application, this is either an I²C bus, an SPI interface, or the parallel interface of the chip.

3.1.4.4.2 Description

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

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

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

There are two kinds of control messages. Normal messages provide control of applications, while system messages handle system-related operations such as resource handling or remote access. During remote access, additional handshaking guarantees reliable remote operation. 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)
8-25	Data area+1 byte message type
26-27	CRC
28-29	Transmission status
30-31	Reserved

Table 3-4: Structure of a control data frame.

Please note:

The contents of register bXTIM – which controls the delay time between two messages in case of low level retries – must be identical in all nodes of a MOST Network.

One complete message is buffered by the chip. Any important status changes can be flagged by interrupt (if desired). The data on the control channel is protected by CRC and acknowledge mechanisms.

Arbitration is provided automatically by the transceiver 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 register bXRTY. If the maximum of retries is reached without success, a transmission error is indicated to the controlling device (e.g., external Microcontroller). The delay time between the sending out of a message (in case of low level retries only) is specified in register bXTIM. The contents of bXTIM must be equal in every node of a MOST Network.

In standalone mode, the MOST control messages can be used to indicate the occurrence of an interrupt.

3.1.5 Internal Services

3.1.5.1 Addressing

The MOST_{Transceiver} supports four different ways of addressing:

- Node position in the ring.
The node position is generated automatically in each node during the locking procedure of the MOST_{Network}.
- Unique node address (2 bytes).
This address can be set by the application. The SAI procedure helps to test on the chip level whether a desired address is unique or not.
- 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_{Transceiver}:

Address range	Mode
0x0001..0x02FF 0x0500..0xFFFF	Normal addressing (Point to point) based on unique node address.
0x0400..0x04FF	Node position addressing: Address = 0x400 + Node position of target node
0x0300..0x03C7 0x03C9..0x03FF	Group addressed: Address = 0x300 + Address of desired group Group address 0xC8 reserved for broadcast
0x03C8	Broadcast addressing

Table 3-5: Addressing modes vs. address range.

Group addressing is typically used for controlling several devices of the same type (e.g., active speakers). The grouping of devices must be established during definition of the system.

3.1.5.2 Address Initialization (SAI)

The MOST_{Transceiver} supports a procedure on the chip level that can be used to verify whether an address is unique or not. The desired address must be written as a target address to the Xmit control message buffer (bytes XTAH and XTAL), and then the bit SAI in the message control register must be set to '1'. Now the transceiver checks whether the address is unique or not. If it is ok, then the address is immediately written to the node address registers of the transceiver. The result of the operation is written to bit TXR in the message status register, with a value of '1' meaning that the address was ok.

3.1.5.3 Support at System Startup

The MOST_{Transceiver} meets all requirements of a low level startup. It is so far able to run in standalone mode, as it provides a communication-ready network to all applications. In addition, several supporting mechanisms are provided. All components of the system get a unique number, with numbering starting at the timing master at 0x00, and then incremented by one. These numbers can be used for node position addressing. Furthermore, every device receives the information about the total number of devices in the ring. The MOST_{Transceiver} also provides a wakeup mechanism.

3.1.5.4 Delay Recognition

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

Therefore a mechanism is implemented in each MOST_{Transceiver}, giving access to information about the total delay of the system, and to the delay up to the local node with respect to the timing master.

3.1.5.5 Remote-Access

This feature provides a remote-controllable I²C bus in each MOST_{Transceiver} that runs in standalone mode. It can be controlled from any node in the network, by using certain system commands. Furthermore, all registers on page 0x0 of the MOST_{Transceiver} can be read or written via the network.

3.1.5.6 Automatic Channel Allocation

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

Allocating one or more audio channels (up to 64 bits per allocation procedure) is done via a request from an application to the timing master of the network. If there are enough channels, the application will get a handle, by which source data can be routed onto the network. The handle can also be used for de-allocating. A channel resource allocation table (mCRA), distributed automatically in the ring (on chip 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 mCRA, or it can be delivered during connection management via control messages. The entire allocation takes about 25ms (maximum), making it possible to change allocation during runtime.

3.1.5.7 Power Management

The MOST_{Transceiver} has three power states:

- Normal operation mode
- Low power mode
- Zero power mode

In normal operation mode, all sections of the chip are running and the chip is fully accessible.

Low power mode is used to lower the power consumption of devices that are not in use at the moment. All sections of the chip that handle source data are switched off, source data bypass is active. The chip is visible from the network's point of view, but no communication is possible. Based on the timing master, all components that are in low power mode can be awakened by a wakeup-preamble.

On zero power mode, all sections of the chip are deactivated, except a small wakeup logic. The wakeup logic activates the receiving FOT unit by a special pin of the chip in regular intervals and scans for modulated light. If there is modulated light, the chip wakes up.

When using an FOT unit with wakeup feature, the zero power mode of the chip does not need to be used. The chip can be connected to the switched branch of the power supply.

3.1.5.8 Detection of Unused Channels

Detection of unused channels, i.e., channels that are allocated by a device, but which are no longer used, is done with the help of the allocation table mCRA. By checking the most significant bit, it can be determined whether a channel is in use or not. If there are unused but allocated channels, they should be de-allocated with respect to the network resources.

3.2 Dynamic Behavior of a Device

3.2.1 Overview

This section describes the dynamic behavior of the system—the states and state transitions of the system, with a special focus on network dynamics (or the dynamic of the network interface of a device). The expression NetInterface stands for the entire communication section of a device, that is, the optical interface, MOST_{Transceiver}, and NetServices.

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_{Transceiver}, the optical interface, and the controller on which the NetServices are running. The NetServices make up the next layer, on which the higher services of address management, power management and network error management are based. At the top layer there is the application itself.

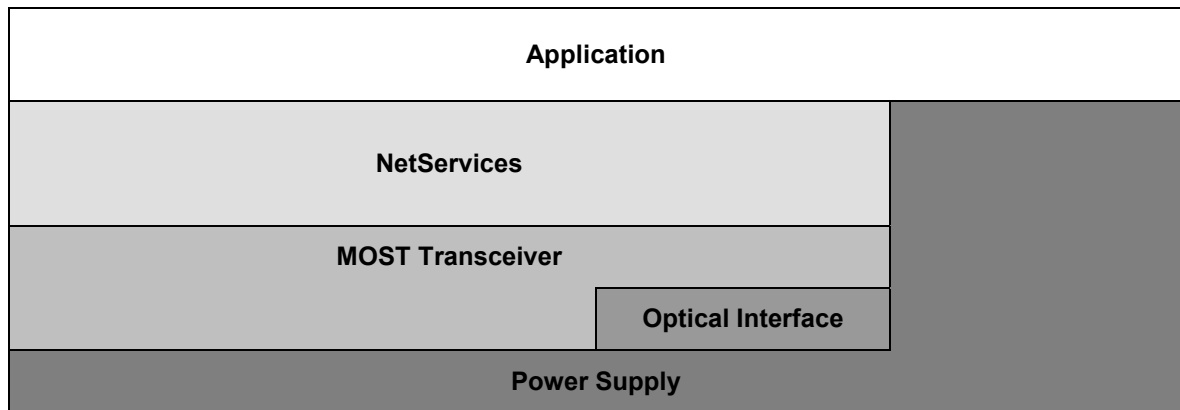


Figure 3-2: Layer model of a device

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

1. **DevicePowerOff:** Communication section is in state NetInterfacePowerOff. The application section in a non-waking device is in state ApplicationPowerOff, or in a waking device in state ApplicationSleep.
2. **DeviceStandBy:** This state is mainly influenced by state ApplicationLogicOnly. The logical function of the application is running, while peripherals with high power consumption such as drives are switched off. This state is reached after state DevicePowerOff. The communication section is in state NetInterfaceNormalOperation.
3. **DeviceNormalOperation:** The communication section as well as the application section are in state NormalOperation.

Since these main states are only a few of all possible device states, it is not useful to use a diagram. The following description gives an impression of what might happen in the single states with respect to the communication and application sections.

DevicePowerOff:

- The application might be awakened, e.g., by a timer, can check an external signal, and return to state ApplicationSleep without waking up the NetInterface. The device does not leave the mode.
- The application might be awakened, e.g., by a timer, and can then wake up the NetInterface, and by that the entire network. The device changes to state DeviceStandBy, or state DeviceNormalOperation
- The application might be awakened by light on the bus and then wakes the application during initialization phase. The device changes to state DeviceStandBy.

DeviceStandBy:

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

DeviceNormalOperation:

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

The following description of the dynamic behavior is done from the bottom up. The most significant subjects regarding power supply are described in section 4.1 on page 161. The following section focuses on the dynamic behavior of the NetInterface.

3.2.2 NetInterface

Here, the states of a device are seen from the view of the NetInterface. Operations within the application of a device are not considered. Only the interfaces to the application are shown. The following figure shows the states of the NetInterface and the events that lead to state transitions. The following sections explain the individual states.

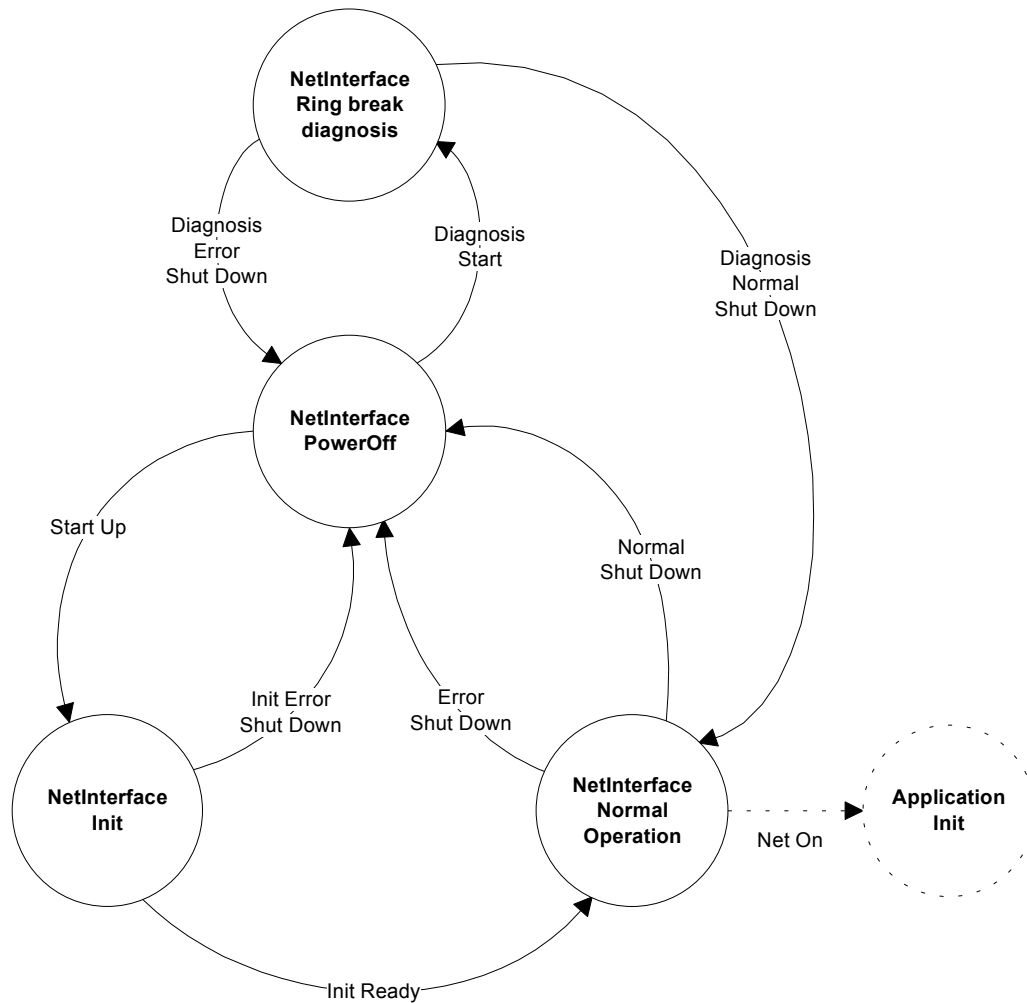


Figure 3-3: Flow chart "Overview of the states in NetInterface"

3.2.2.1 NetInterfacePowerOff

In state NetInterfacePowerOff, the NetInterface is switched off from the view of the network. The FOT does not emit light. The MOSTTransceiver does not necessarily need to be switched off, since the application might still use function groups of the chip (e.g., RMCK generation).

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

Event	Transition to	Cause
Start Up	NetInterfaceInit	A NetInterface is activated either by light at the receiving FOT, by the application (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 161)

Table 3-6: Events in state NetInterfacePowerOff

3.2.2.2 NetInterfaceInit

In this state, NetInterface is initialized to the point where the MOSTTransceiver is able to communicate with other nodes.

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

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

Table 3-7: Events in state NetInterfaceInit

Causes for event Init Ready:

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

Causes for event Init Error Shut Down:

- In the master device:
Timeout t_{Master} , occurs before a stable lock can be recognized.
- In a waking slave device:
Timeout t_{Slave} , 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_{Slave} , expires before light was recognized, or a closed ring was recognized; or the light was switched off again.

In a slave device (non-waking) the bypass of the transceiver 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 (light at the output).

The flow chart below shows the behavior in state NetInterfacelnit. A differentiation is made between master and slave. On this level, master means timing master and slave means timing slave.

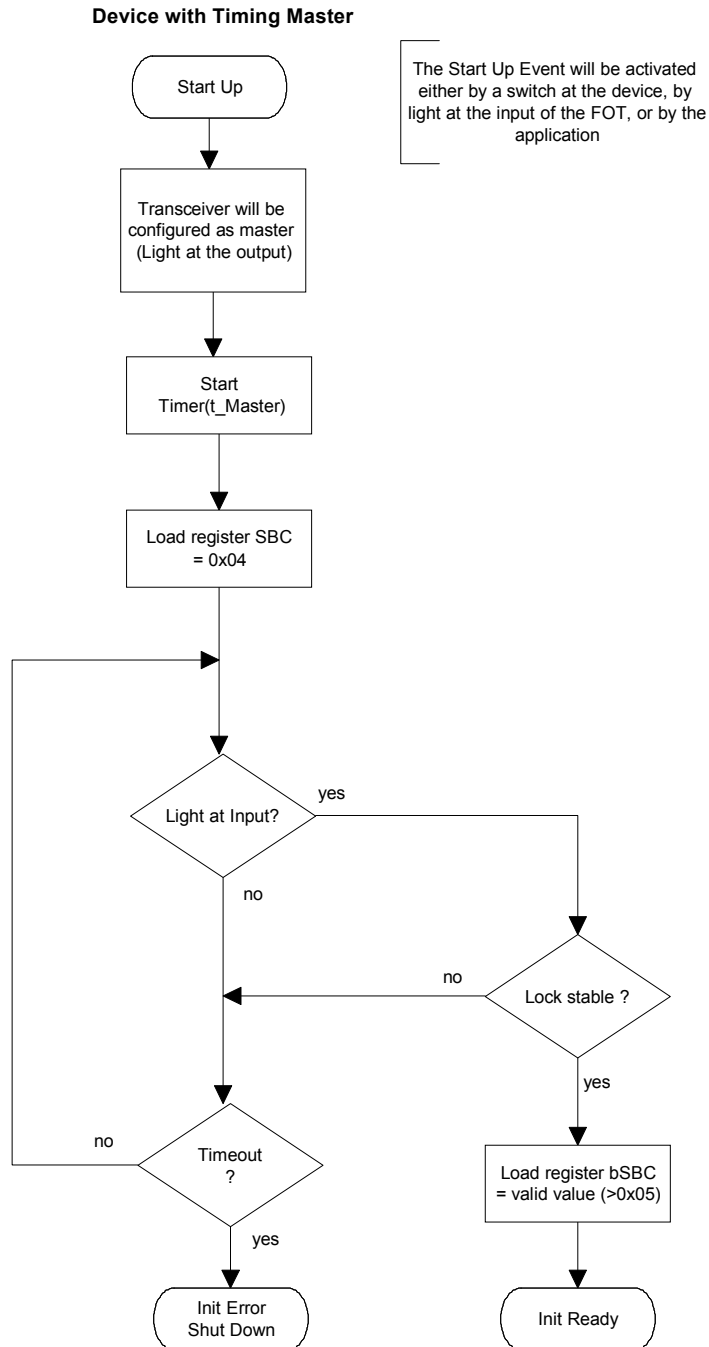


Figure 3-4: Behavior of a master device in state NetInterfacelnit.

When entering state NetInterfacelnit, the timing master loads its bSBC register with the “invalid” value 0x04. This value is transferred to all transceivers via the frame. As soon as the timing master recognizes a stable lock, it loads its bSBC register with a valid value (>0x05). By doing this, every slave in the ring can recognize when the timing master has reached stable lock.

Woken Slave Device

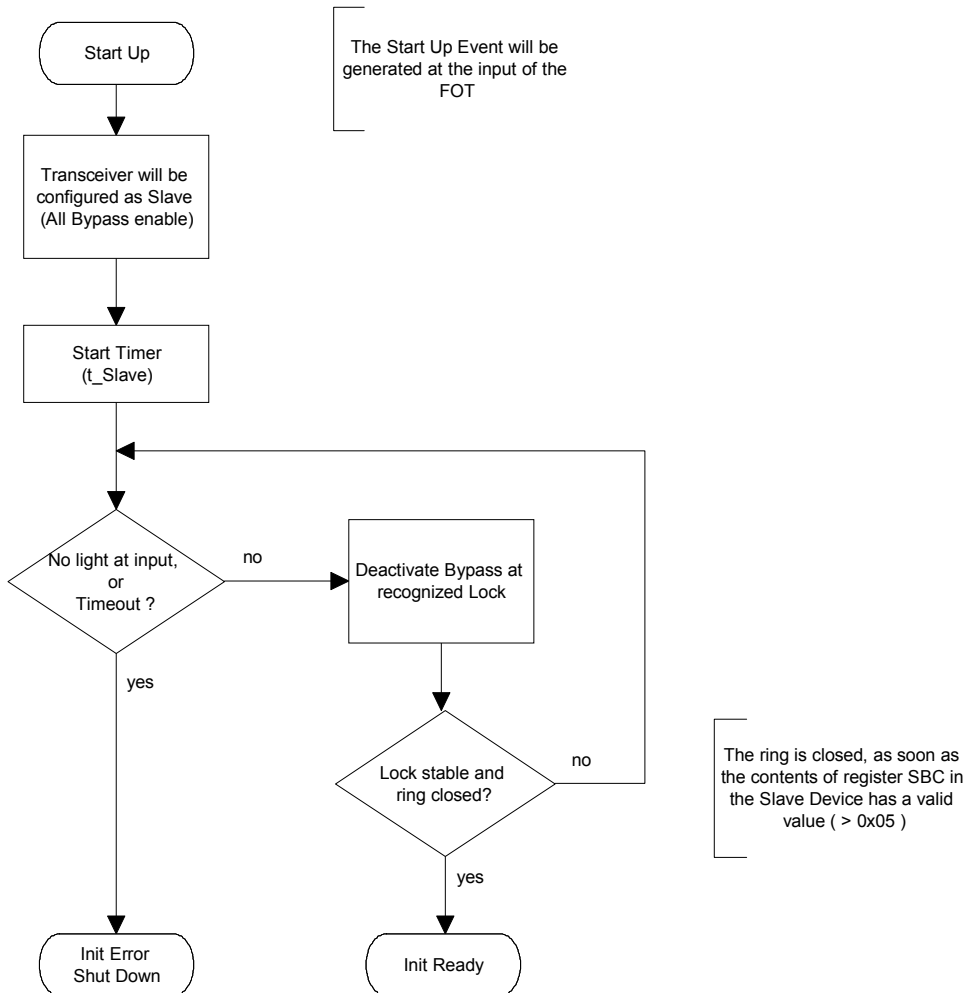


Figure 3-5: Behavior of a waked slave device in state NetInterfacelnit.

Waking Slave Device

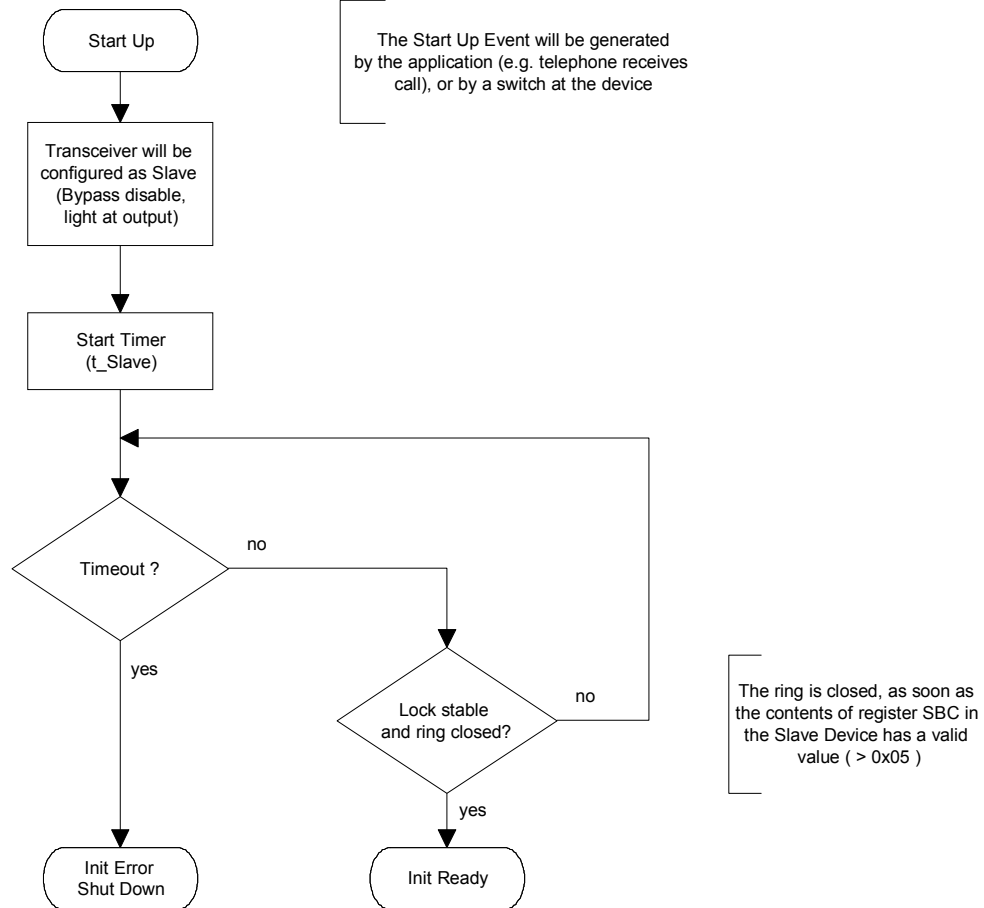


Figure 3-6: Behavior of a waking slave device in state *NetInterfaceInit*.

3.2.2.3 NetInterfaceNormalOperation

This state is reached as soon as the initialization has reached a level where the MOST_{Transceiver} 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 Net On Event:

- Check of system configuration and building of the central registry (refer to section 3.3.5.3 on page 131).
- Setting of the logical node address and group address (refer to section 3.3.1 on page 127).
- Initialization of the sending and receiving units in the NetServices.

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

Event	Transition to	Cause
Normal Shut Down	NetInterfacePowerOff	NetInterface will be deactivated by switching off light.
Error Shut Down	NetInterfacePowerOff	NetInterface will be deactivated due to a critical unlock.
Net On	Report to an application	Entering state NetInterface Normal Operation

Table 3-8: Events in state NetInterfaceNormalOperation

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

In state NetInterfaceNormalOperation, the NetServices check the lock state of the PLL of the MOST_{Transceiver}. On an unlock, the application is informed as soon as possible by an unlock event. Every application then has to save its output signals (e.g., amplifier mutes its outputs).

In addition to that, the NetServices check 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.

If there are no further unlocks within time $t_{UnlockRecovery}$ before t_{Unlock} is exceeded, the lock state is regarded as stable. The application will get informed about this Lock Event, for restoring its signals.

In case a series of unlocks occurs, where the single unlocks do not exceed t_{Unlock} , but the interval of the occurrence is less than $t_{UnlockRecovery}$, an Error Shut Down event (critical unlock) is generated as well.

In addition to that, the MPR register is checked. If this register indicates a NetworkChange event, the application will get informed about that.

The flow chart below shows the behavior in state NetInterfaceNormalOperation:

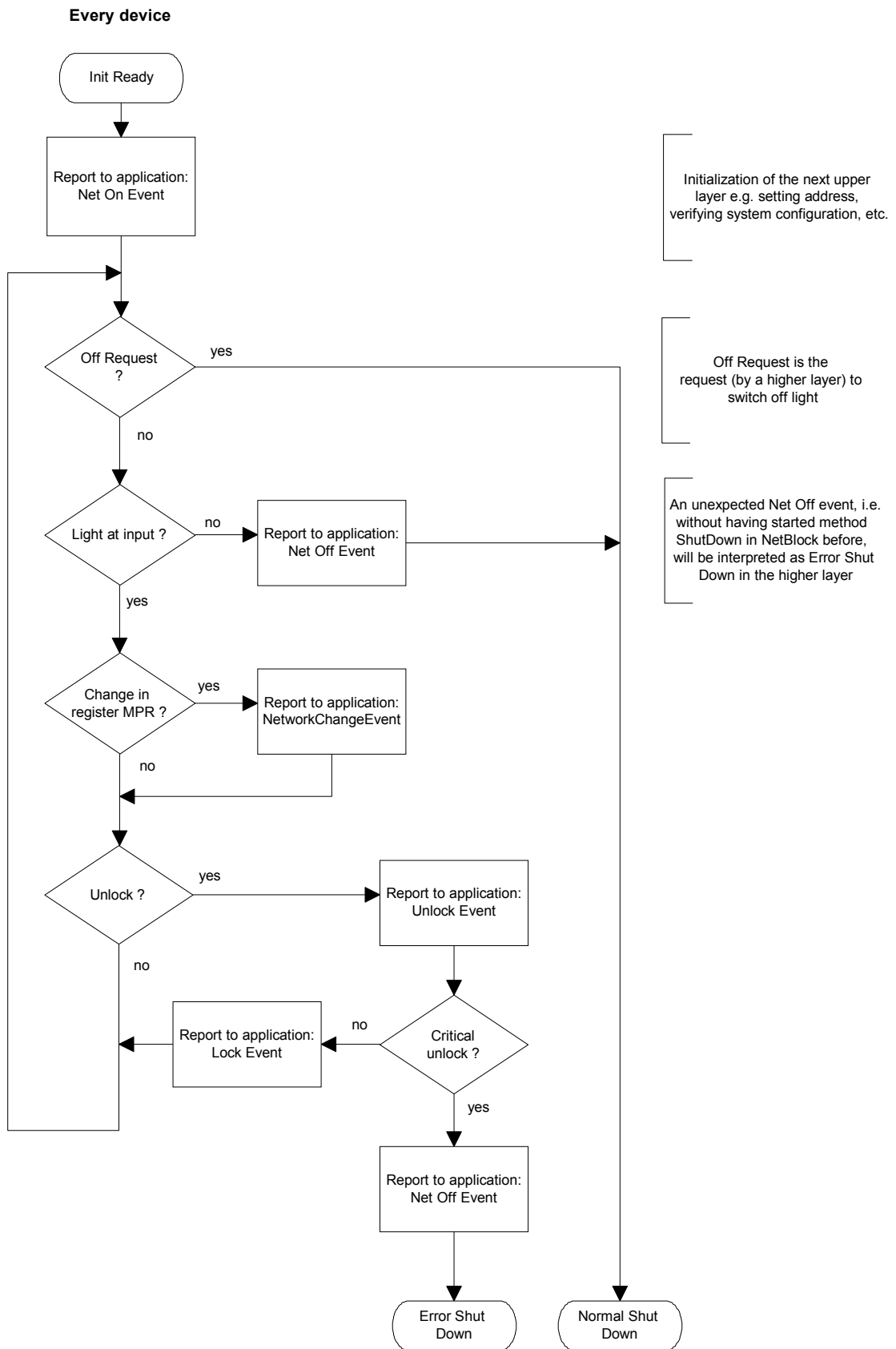


Figure 3-7: Behavior in state NetInterfaceNormalOperation.

3.2.2.4 NetInterface Ring Break Diagnosis

A simple recognition of a fatal error is possible in any state. Ring break diagnosis serves the purpose of localizing a fatal error in the network. It is run not during normal operation, but in the car repair, or at the assembly line.

RingBreakDiagnosis is entered, when signal SwitchToPower of the SwitchToPowerDetector indicates, that the device was connected to power first time (e.g. after reconnection of the car's battery). This signal is not evaluated during NetOn.

In state NetInterfaceRingBreakDiagnosis the network cannot reach normal operation. In this state, a relative node position is determined in every device. This information can be used in case of a fatal error (ring break or defective device) to localize the error.

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

In case of a Diagnosis Error Shut Down event, the position determined in each device describes the position relative to the device that was configured as timing master at the end of RingBreakDiagnosis (since there was no light at its input).

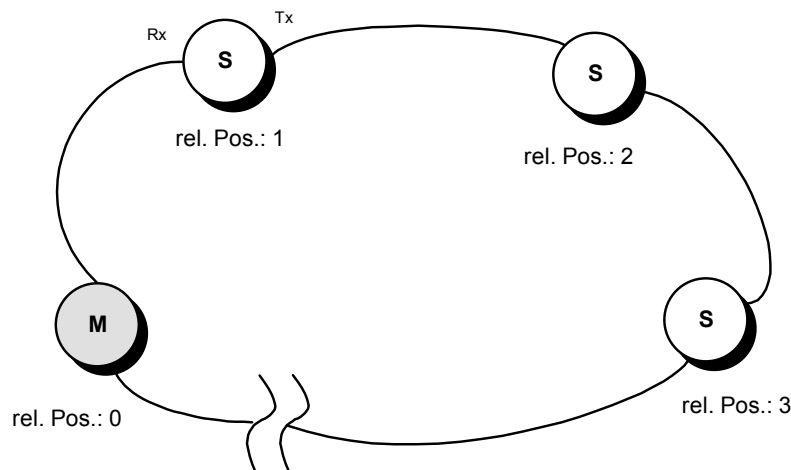


Figure 3-8: Localizing a fatal error with the help of ring break diagnosis.

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

Table 3-9: Events in state NetInterfaceRingBreakDiagnosis

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

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

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

If, on stable lock, the bSBC register contains a value greater than 5, the ring is closed. There is no defect and the NetInterface changes to state NetInterfaceNormalOperation.

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

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

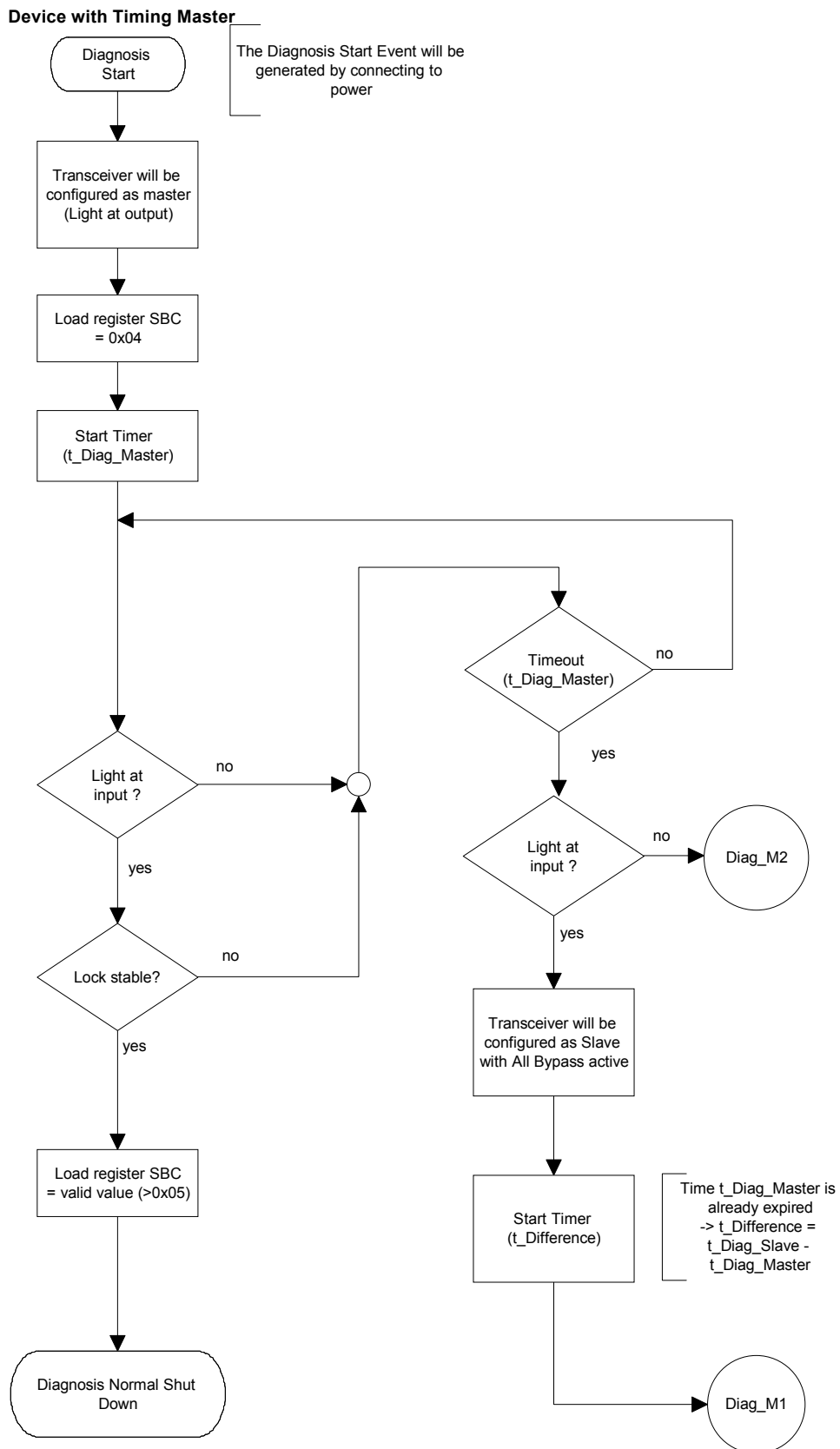


Figure 3-9: Behavior during ring break diagnosis in a timing master (part 1).

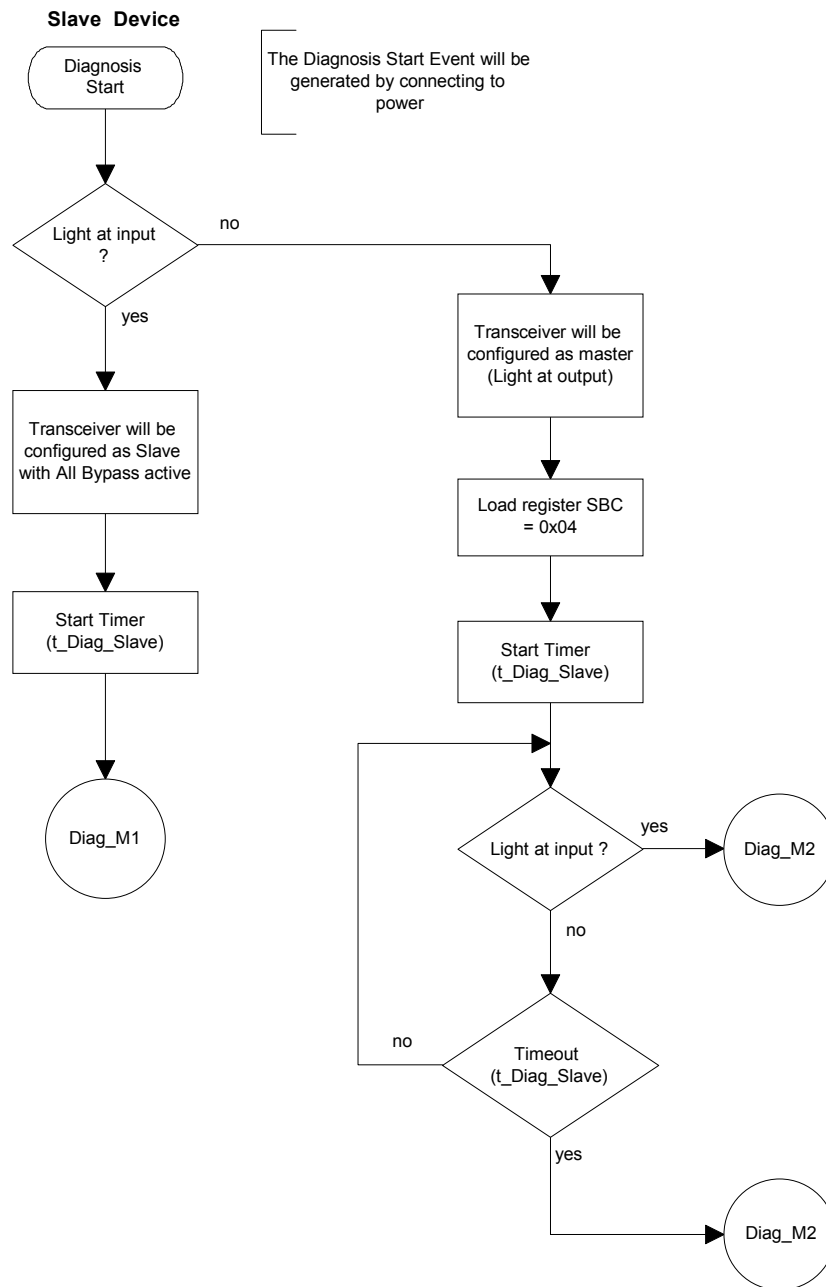


Figure 3-10: Behavior during ring break diagnosis in a slave (part 1).

Every Device

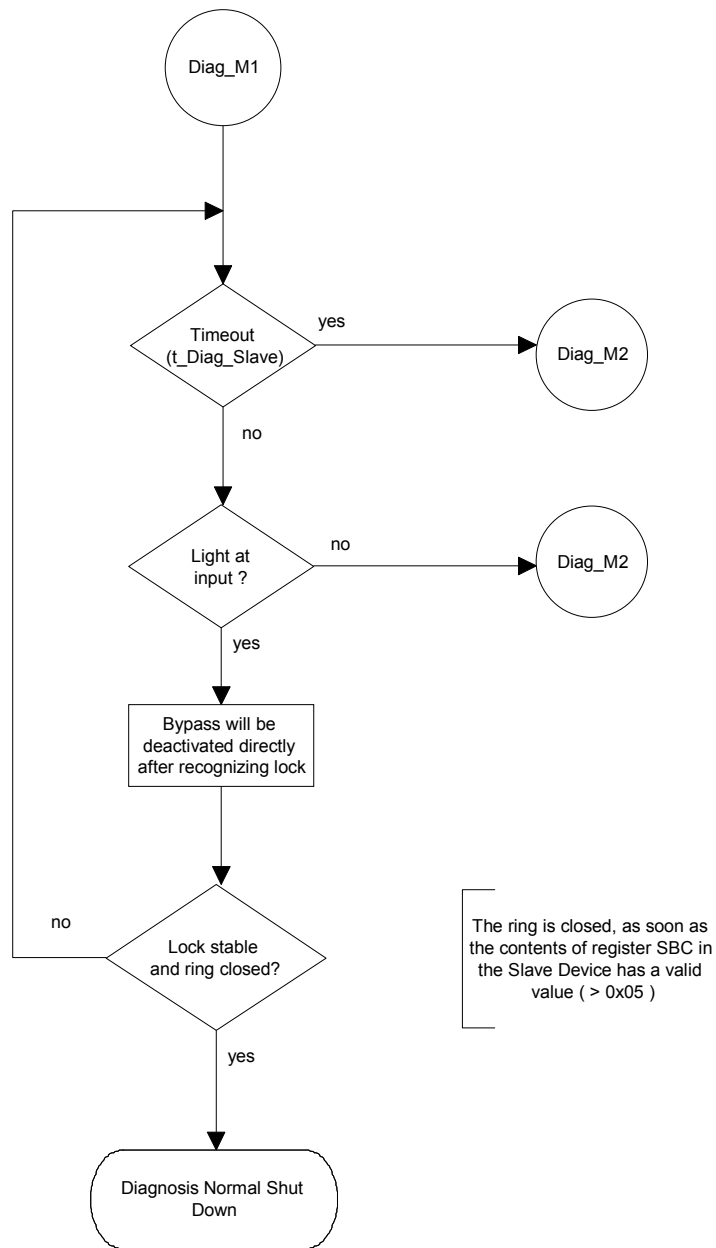


Figure 3-11: Behavior during ring break diagnosis in a timing master and slave (part 2).

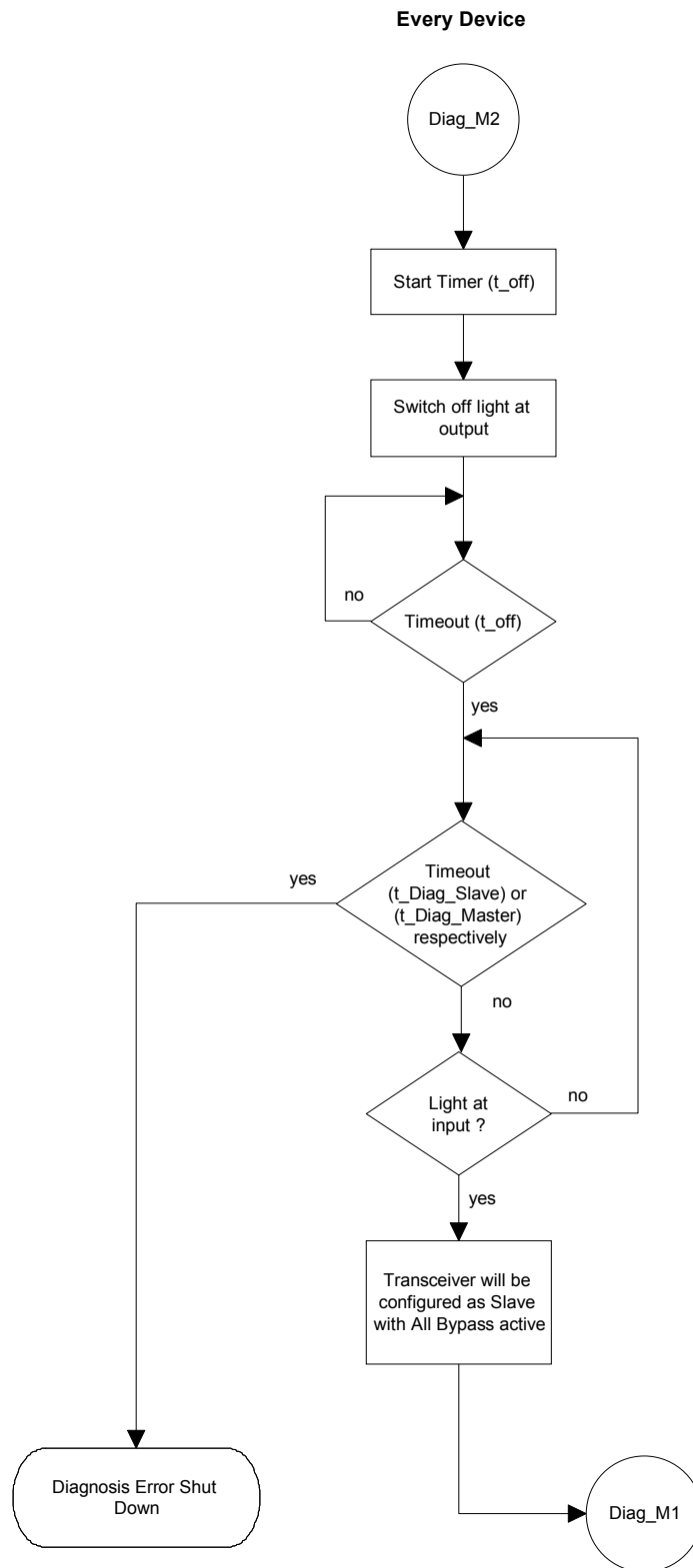


Figure 3-12: Behavior during ring break diagnosis in a timing master and slave (part 3).

3.2.3 Initialization on Application Level

With the NetOn event, the NetInterface of a single device shows its local application readiness for communication. The network on a basic level is now ready for communication. Initializations can be performed that have to do with the interacting of multiple devices on the application layer, and the system (this does not mean that initialization of the individual applications shall start at this time and not earlier). The network master checks system configuration. Depending on the result, the logical node addresses are initialized. Then the application can initialize the communication controlled by itself, i.e., it registers in the notification matrices. Initialization can be named "SystemCommunicationInit".

As a complicating factor, the system must be prepared for devices connecting to or disconnecting from the network (Network Change Event). In these cases, the system must run consistently without disturbances, and re-initializing phases must be as short as possible. On a NetworkChange event, parts of SystemCommunicationInit must be run again, but initialization must not be run completely due to the time this would take.

3.2.3.1 Network Slave

The flow after a NetOn event in a network slave is displayed in the following diagram. At first, the slave writes the logical address, which was stored during the last run of the system, to the MOSTTransceiver. If it has no stored address, for example, if it lost power or it did not yet run in the network, it writes 0xFFFF to the MOSTTransceiver. The slave then waits for Broadcast Configuration.Status of the network master. While it waits, it responds to requests, for example, of FBlockIDs by the network master.

If Configuration.Status (OK) is received, the slave takes its de-central registry. On Configuration.Status (NotOK), the slave rejects its de-central registry and the logical node address of the last system run. It determines the new logical node address from the node position and writes it to the MOSTTransceiver. The de-central registry is rebuilt if required. On a regular ShutDown, that is, after a ShutDown.Start, each device stores its de-central registry and its logical address in buffered RAM. If a device is removed from power, it has to delete the de-central registry and its logical address.

Please note:

A Slave must not store the logical address of the Network Master, and must not assume 0x0100 as default address. The logical address of the Network Master must be derived from Broadcast.Configuration.Status.

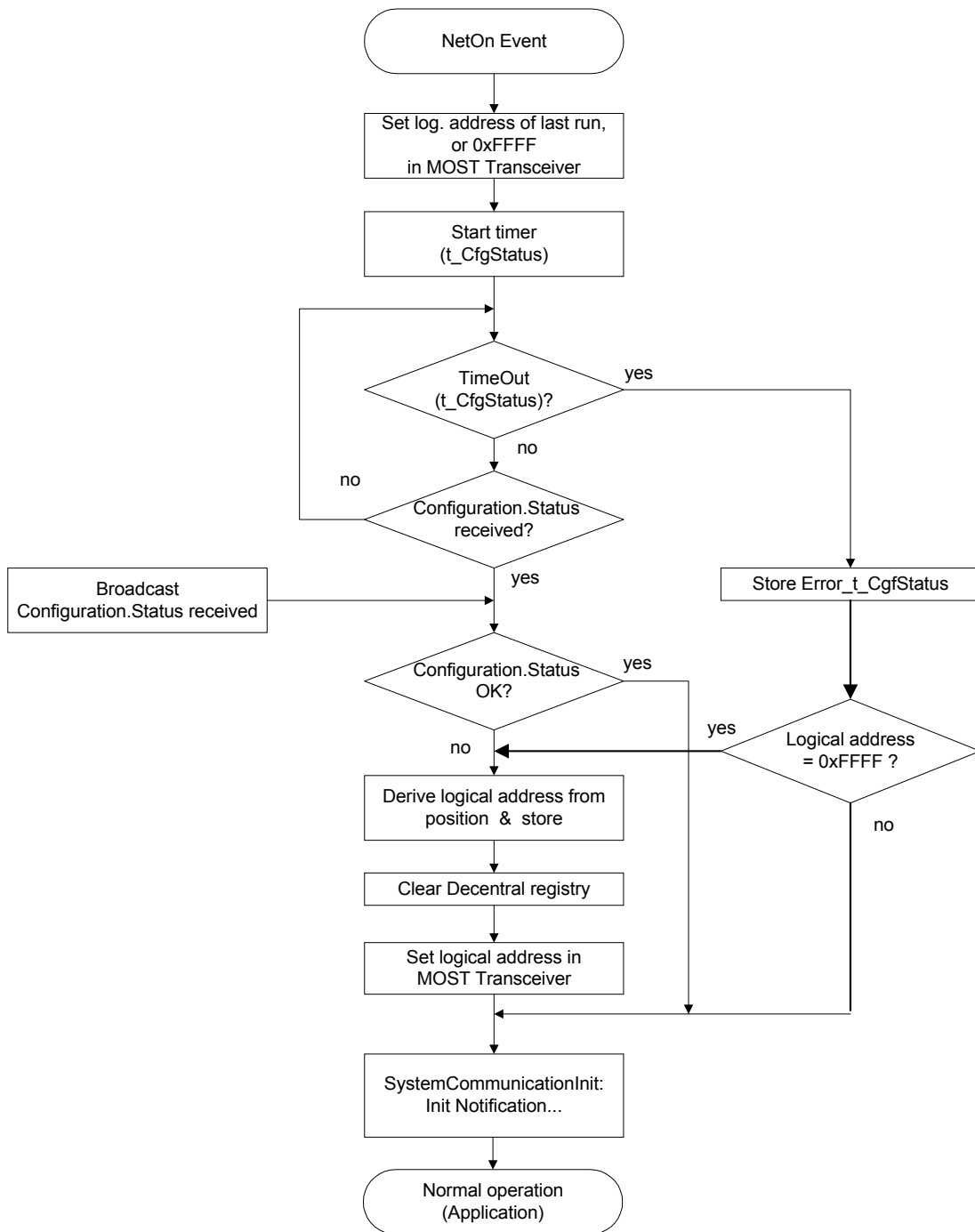


Figure 3-13: Flow of initialization on application level in a network slave.

Now the communication initialization of the application can start. During SystemCommunicationInit, the notification is built up (beneath others). Here, the application of the device registers in the notification matrices of function blocks whose properties are relevant for it.

It may happen that a device does not become active in the network, or that a device becomes inactive. Every application must therefore be able to handle a missing communication partner, to initialize and run dependent parts of the application adapted to the circumstances, or even to omit some parts.

An example could be the initialization of the notification in an MMI. For this example, assume that it controls an AM/FMTuner, an AudioDiskPlayer and a NavigationSystem. At first, only the AM/FMTuner and AudioDiskPlayer are available in the system. During the first run of SystemCommunicationInit, the MMI registers in the notification matrices of AM/FMTuner and AudioDiskPlayer. After that, it receives their current status reports. The control and displaying elements of the Navigation System will turn to gray. The rest of the system runs without the NavigationSystem.

It may happen that a device connects late. This is indicated by a NetworkChange event. After such an event, the system configuration must be checked, since it might be that the new device is not initialized (Rx/TxLog=0xFFFF). It may also be necessary to run SystemCommunicationInit (or parts of it) again. Therefore a jump in the initialization flow is performed, which is triggered by the reception of Configuration.Status from NetworkMaster.

The example above might be continued by the NavigationSystem entering the network. The network master checks system configuration then and broadcasts Configuration.Status (OK), since the NavigationSystem is already known and initialized. For achieving a fast startup, the MMI should recognize that the initialization of AM/FMTuner and AudioDiskPlayer has already been done. Then it needs to initialize the navigation system only.

3.2.3.2 Network Master

The network master first checks to see if it has a logical node address stored in buffered RAM. If not, that means it was removed from power, and it assumes that the entire system must be initialized completely. It broadcasts Configuration.Status (NotOK), then determines its logical address based on its position.

If the network master finds a valid node address (e.g. 0x0100) in its RAM, it writes it to the MOSTTransceiver and starts checking the system configuration. (This is described in detail below.) If the logical addresses of the currently-available devices match the entries of the central registry from the last system run, and if no device has an uninitialized address (0xFFFF), the network master broadcasts network master Configuration.Status (OK). By doing this, the network master tolerates devices that are now missing but were present in the old registry, since those devices may become active later.

A NetworkChange event leads to a jump to the initialization on the application level, so the network master can check the new system configuration.

Only in case of a system shutdown in proper form (as a result of ShutDown.Start), the differences to the reference configuration are written to the error memory (Error_Registry_New).

Example:

1. Starting situation:

Desired configuration:

Rx/TxLog	FBlockID	InstID
0x0101	AudioDiskPlayer	1
0x0102	AM/FMTuner	0
	AudioDiskPlayer	2
0x0103	TVTuner	0
0x0104	AudioAmplifier	1
	AudioAmplifier	2

central registry of last system run

Rx/TxLog	FBlockID	InstID	available?	Position
0x0101	AudioDiskPlayer	1		
0x0102	AM/FMTuner	0		
	AudioDiskPlayer	2		
0x0103	TVTuner	0		
0x0104	AudioAmplifier	1		
	AudioAmplifier	2		

2. System startup - Checking configuration: Devices 0x0102 and 0x0103 available

Rx/TxLog	FBlockID	InstID	available?	Position
0x0101	AudioDiskPlayer	1	no	
0x0102	AM/FMTuner	0	yes	1
	AudioDiskPlayer	2		
0x0103	TVTuner	0	yes	2
0x0104	AudioAmplifier	1	no	
	AudioAmplifier	2		

⇒ Broadcast Config.Status (OK)

3. Late activation: Device 0x0104 joins network; Checking configuration

Rx/TxLog	FBlockID	InstID	available?	Position
0x0101	AudioDiskPlayer	1	no	
0x0102	AM/FMTuner	0	yes	1
	AudioDiskPlayer	2		
0x0103	TVTuner	0	yes	2
0x0104	AudioAmplifier	1	yes	3
	AudioAmplifier	2		

⇒ Broadcast Config.Status (OK)

4. ShutDown: Normal ShutDown with ShutDown.Start; Store error "Device 0x0101, AudioDiskPlayer.1 missed"

3. Variant: uninitialized device 0xFFFF joins network at node position 2; Checking configuration; Recognizing uninitialized device in system; Broadcast Config.Status (NotOK); Building addresses; Building central registry

Rx/TxLog	FBlockID	InstID	available?	Position
0x0101	AudioDiskPlayer	1	no	
0x0102	AM/FMTuner	0	yes	1
	AudioDiskPlayer	2		
0x0103	TVTuner	0	yes	3
0x0104	AudioAmplifier	1	yes	4
	AudioAmplifier	2		
0x0102	Speech Recognition	0	yes	2

⇒ Broadcast Config.Status (OK)

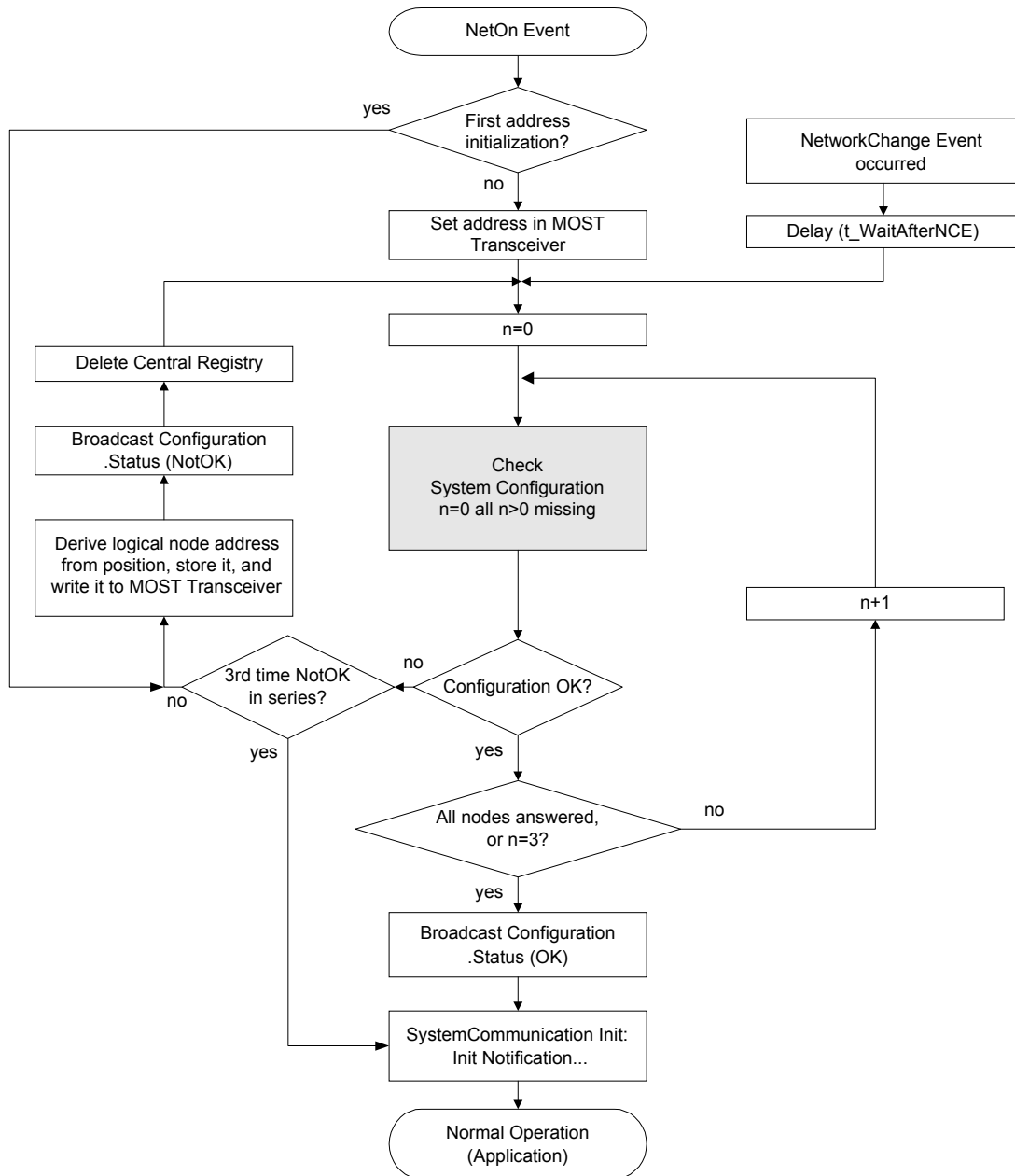


Figure 3-14: Flow of initialization on application level in a network master.

For checking system configuration (Figure 3-15 on page 116), the network master requests the FBlockIDs from each node in the network, using node position addressing. The logical node address is contained in the answer given by the addressed node, and is set in the MOSTTransceiver. If the network master recognizes an uninitialized device (Rx/TxLog=0xFFFF), the system configuration check is interrupted by Configuration NotOK. This also happens if two devices try to use the same address, or the assignment function blocks <-> logical address is no longer correct. If all devices are requested and their data corresponds with the entries in the stored central registry (logical address as well as contained function blocks), the process finishes with Configuration OK.

If one or more devices did not respond on the request of FBlockIDs, it is tried – after timeout - up to two times to request FBlockIDs again. The number of devices is derived from the MPR register of the MOST Transceiver. If the Network Master tried to check system configuration the third time in direct succession, and if it was not ok, checking is cancelled.

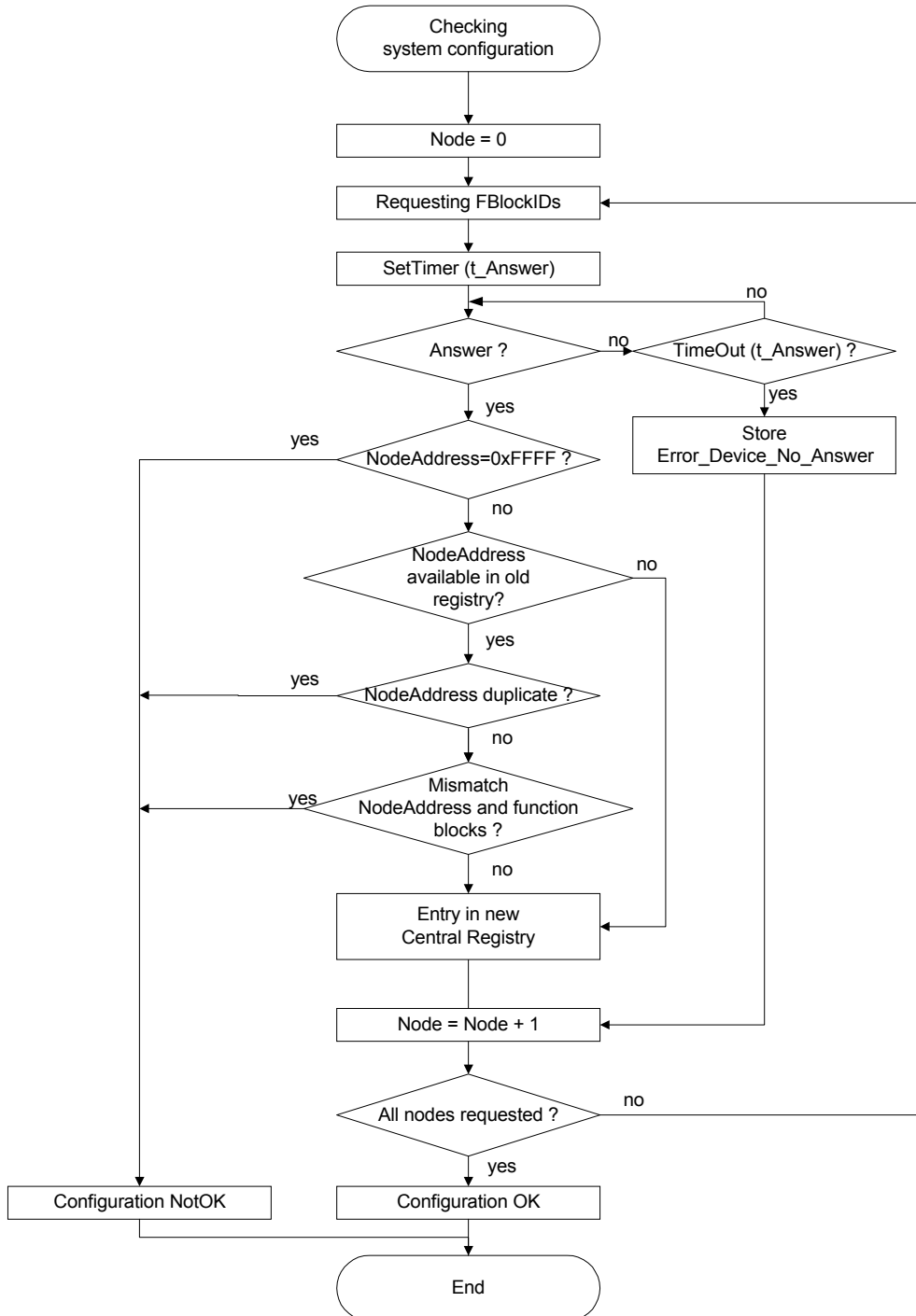


Figure 3-15: Flow in network master during requesting system configuration.

The flow in Figure 3-15 is shown in serial for clarification. The NetworkMaster should use a parallel approach, i.e. requesting several devices, and no waiting for answers only.

3.2.4 Power Management

3.2.4.1 General Procedure

Power management means that the administrative function, which is above the NetServices, wakes and shuts down the MOSTNetwork. The power management is handled mainly by the function block power master.

- **Waking of the network:** Waking the network is done by emitting modulated light (light on). In principle the network can be awakened by any node. The ability of a node to wake up the network can be activated or deactivated by the power master (e.g. in case of a critical charge status of the accumulator) in the property **AbilityToWake**, which is implemented in every NetBlock. The power master itself will usually wake the network, for example, when there is communication on the car's bus, or based on the status of the vehicle (Clamp status).

Please note:

A device must only wake the network when this is initiated by the application. Failure (e.g., supply voltage too low, or too high) must not initiate waking of the network.

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

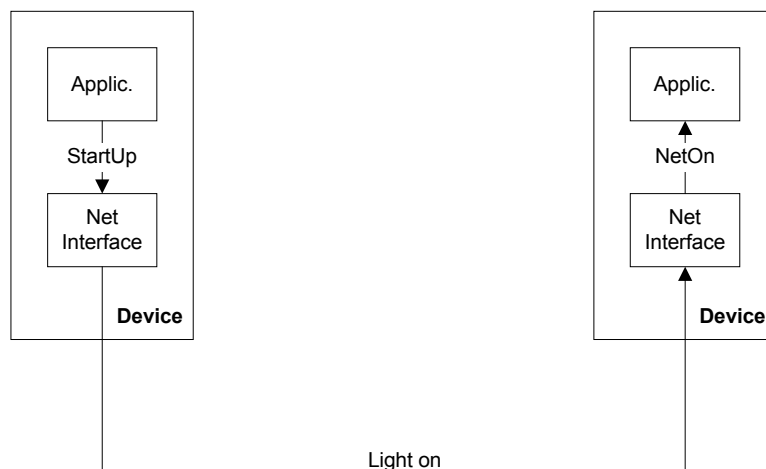


Figure 3-16: Example (2 devices) for waking of the MOST network via light on the network.

- **Switching off network:** Switching off the network is done on lowest level by switching off light. A device, which has switched off light, must not switch it on again before t_{Restart} occurred. This applies also to that case, if it recognizes light at its input.

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

In all other cases, only the power master switches off the network. For avoiding that devices have to save their status to non-volatile RAM very often, the Power Master 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 power master 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 Power Master by starting ShutDown.Start(Execute).

By this function call, the shutdown process is started irrevocably. Every device has to prepare for shutting down without reply to the PowerMaster (Saving status) and waits for the light to be switched off.

The PowerMaster switches off light $t_{ShutDownWait}$ after ShutDown.Start(Execute). This time allows to shutdown audio output without audible side effects.

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

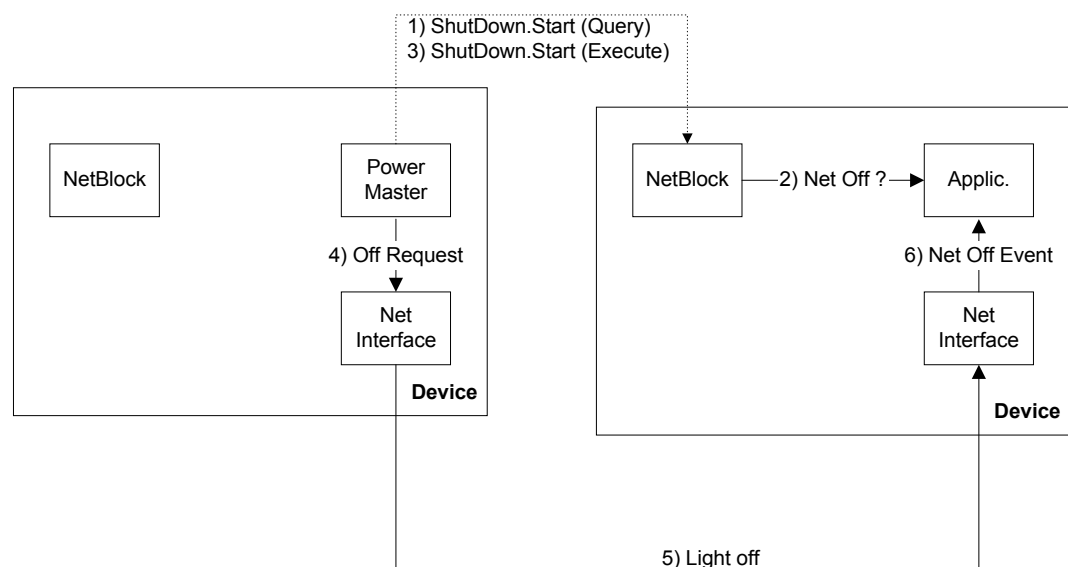


Figure 3-17: Switching off MOST network via starting method ShutDown in every NetBlock, and signaling to every application, and switching off light.

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

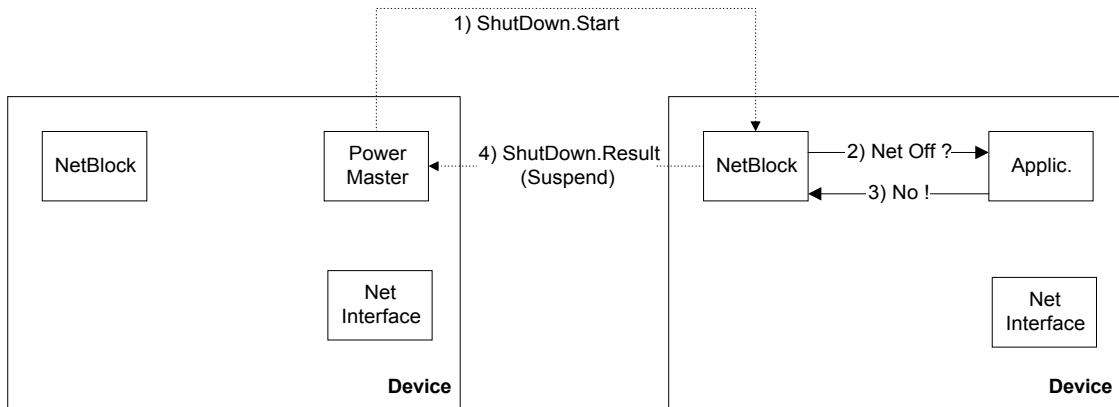


Figure 3-18: Prevention of switching off MOST network via ShutDown.Result (Suspend)

Please note:

If the light is switched off during startup, e.g. by low voltage, long unlock or fatal error, the PowerMaster must not wake the network for being able to finish its shutdown procedure. It will abort its startup procedure.

FUNCTIONS				
FktID	OPType	Sender	Receiver	Explanation
AbilityToWake	Set	Power Master	one NetBlock	On: Device can wake the network Off: Device must not wake the network
ShutDown	Start	Power Master	all NetBlocks	Announcing of switching off network
	Result	one NetBlock	Power Master	Reply on further demand for communication

Table 3-10: Functions in conjunction with power management.

3.2.4.2 Functions and Important Operations

The functions described above and the related operations are explained below with the help of the respective protocols.

AbilityToWake:

```
??? -> Slave: NetBlock.Pos.AbilityToWake.Set (WakeStatus)
```

WakeStatus	0x00	Off (Default for a non-waking device)
	0x01	On (Default for a waking device)
	0x02	Critical

```
??? -> Slave: NetBlock.Pos.AbilityToWake.Get  
Slave -> ??? : NetBlock.Pos.AbilityToWake.Status (WakeStatus)
```

ShutDown:

Since the InstID is ignored in NetBlock, 0xFF must be regarded as a dummy value.

```
??? -> all: NetBlock.FF.ShutDown.Start (Query)
```

```
??? -> all: NetBlock.FF.ShutDown.Start (Execute)
```

```
Slave -> ??? : NetBlock.Pos.ShutDown.Result (Suspend)
```

Query	0x00
Suspend	0x01
Execute	0x02

3.2.5 Error Management

In the network the following errors might occur on the lowest level:

- **Fatal Error:** Error that leads to the interruption of the ring, to the breakdown of the network, or that means the network can not be initialized (Super- or sub-voltage, ring break, defect FOT unit).
- **Unlock:** The PLL of the MOST_{Transceiver} 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.
- **Voltage Low:** The voltage of one or more devices is too low to maintain operation of the NetInterface.

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

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

3.2.5.1 Handling of Light Off

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

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

If light was switched off, it might be the case that it is switched on again after a short time. If the application would shut down immediately, some devices might need a long time to return to normal operation. Therefore the application has to be prepared for ShutDown, but has to stay active for $t_{PwrSwitchOffDelay}$. If the light reappears within $t_{PwrSwitchOffDelay}$, the system is re-initialized like when waking up after sleep mode. The only difference is, that within the devices power supply, micro controller, and operating system need not to be re-initialized.

3.2.5.2 Fatal Error

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

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

3.2.5.2.1 Waking

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

3.2.5.2.2 Operation

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

3.2.5.3 Unlock

An unlock occurs when a timing slave cannot lock onto the input signal of the PLL of the MOSTTransceiver, or if a timing master does not receive a comprehensible signal.

One cause for this might be that two timing masters in one ring work against each other. This case can be recognized only in the timing masters themselves.

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

Reaction of NetServices:

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

Reaction on application level:

The application secures the synchronous signals. An audio amplifier must mute as fast as possible (refer to section 3.7.1.4 on page 158). After a lock is established again (recognized by the NetServices), the application restores its synchronous signals as fast as possible (e.g., de-mute). This must happen only if there is no NetworkChange event, where a node has closed its bypass and therefore has left the network.

Reaction on network masters:

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

3.2.5.4 NetworkChangeEvent

A NetworkChangeEvent occurs, if a device opens or closes its All Bypass, i.e. enters or leaves the network. A NetworkChangeEvent is recognized by the NetServices in every device, by a change of the MPR register in the MOST Transceiver.

If a short unlock occurs during the NetworkChangeEvent, it is caught by the corresponding error handling. If an additional node joined the network, the new node must be integrated on system level. Therefore SystemCommunicationInit must run, but only partially. In addition to that, the network master checks configuration again, and a jump to initialization with Configuration.Status is performed in each device.

If a node has left the network, the output signals that depend on synchronous data transfer must be secured immediately. They must not be restored after the disappearance of the (eventually short) unlock, since it might be that the source of a synchronous signal is missing. Furthermore, every node must be able to handle the case where a communication partner is missing, and must terminate dependent parts of the application in a safe way. Also in this case the NetworkMaster checks system configuration, but without broadcasting Configuration.Status.

3.2.5.5 Low Voltage

For exact values, hysteresis etc., refer to section 4.7 on page 172 please. A too-low supply voltage does not inevitably occur in every device at the same time and in the same intensity. As already described, there are two limits regarding the supply voltage of a device:

Critical voltage U_{Critical} :

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

Low voltage U_{Low} :

There is a second limit, where even the NetInterface no longer works reliably, so even communication cannot be maintained. For being able to survive short intervals of low voltage, a Power Save Mode should be implemented:

Power Save Mode: During the initialization process, the initializing of operation systems, initialization of the application, and the communication takes most of the time, while the actual network is initialized within approximately 100ms. At low voltage the re-initialization of operating systems, the application, and communication must be prevented for as long as possible. Especially in case of devices that have long initialization intervals. Suitable actions to be taken are: buffering of supply voltage, unloading protection, switching off peripherals of the application, resetting of the MOSTTransceiver (closes its all bypass), stopping the controller, etc. For avoiding to disturb communication on the network, light must not be switched off in Power Save Mode. The device containing the timing master, must not close its All Bypass.

If the low voltage cannot be “survived” and the device is reset, then it switches off the light, rejects the initialization states, and switches to normal DevicePowerOff Mode, as if it was switched off. The device stays in DevicePowerOff mode, even if the supply voltage recovers. It is awakened either by “light on” at its input, or by the demand for communication from its own application. It changes to mode DeviceNormalOperation via the standard initialization process. The low voltage reset leads a device to normal behavior.

By opening the 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.

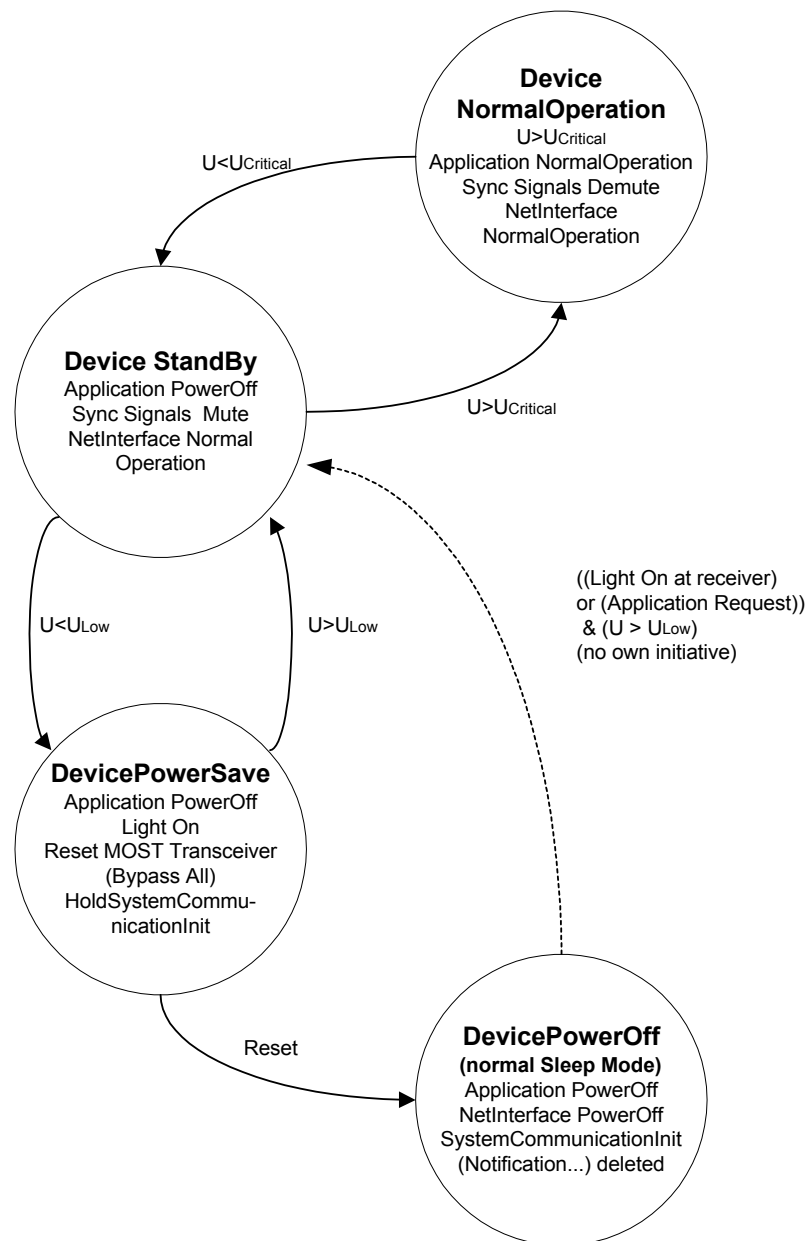


Figure 3-19: Behavior of a device depending on supply voltage.

3.2.5.6 “Hanging” of an Application

By implementing a watchdog in each device, a long “hanging” of the application should be avoided. This effect is reduced to a NetworkChangeEvent (Closing of All Bypass), and an eventual second NetworkChangeEvent (Opening of All Bypass).

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

3.2.5.7 Diagnosis

KWP2000 messages are transported to the FBlocks Diagnosis in all MOST Devices via the MOST Network. Except the transporting, diagnosis of the Devices is handled in the common way. By an identifier (range 0x80 up to 0x9F), different information like serial number, hardware version, software version etc., can be accessed. This range is identical in every device. The identifier ranges between 0x01 up to 0x7F, and between 0xA0 and 0xEF are reserved for device specific functionality's.

Within the “private” identifier range, the values 0xA0 up to 0xAF are reserved for MOST specific functionality's in every device.

ID	Description	Data type which is accessed	Kind of access via KWP2000
0xA0	Version of MOST Transceivers and of MOST NetServices, and revision (Manufacturer specific version) of MOST NetServices	Byte 0-2 Version MOST Transceiver Byte 3-5 Version MOST NetServices Byte 6-8 Revision MOST NetServices	read only
0xA1..0xAE	Reserved		
0xAF	Control of Switch 1 (Optical Interface Area) in all devices, for temporary reduction of optical power.	Byte 0 Bit 0 = 0 S1 open Bit 0 = 1 S1 closed Bit 1..7 reserved	read and write

Table 3-11: KWP2000 on MOST. Overview of identifiers.

The MOST Function DeviceInfo which is part of every NetBlock, uses the same identifiers in the range from 0x80 up to 0xAF for accessing the same information. So DeviceInfo represents a kind of window to information stored in the diagnosis area.

As an example the requesting of the VehicleManufacturerECUHardwareNumber (ID 0x91) of a device at node position Pos is done as follows:

```
NetBlock.Pos.DeviceInfo.Get (ID=0x91)
```

And the answer will be:

```
NetBlock.Pos.DeviceInfo.Status (ID=0x91, VehicleManufacturerECUHardwareNumber)
```

3.3 Accessing Control Channel

3.3.1 Addressing

In a MOSTNetwork, nodes in a ring are addressed. The MOSTTransceiver provides four different types of addresses, which are described below.

- **Node Position Address (Rx/TxPos)**
Physical position of the transceiver in the ring (0x00 up to 0xFF). Counting for this address starts at 0x00 in the timing master node. It is called RxPos for a receiving node, and TxPos for a transmitting node.
- **Logical Node Address (Rx/TxLog)**
User definable address between 0x0001 up to 0x02FF, and 0x0500 up to 0xFFFF. 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. Valid addresses are 0x00 up to 0xC7, and 0xC9 up to 0xFF.
- **Broadcast address (0x03C8)**
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 MOSTTransceiver 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 network master 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 position value:

$$\text{Rx/TxPos} = 0x0400 + \text{Pos}$$

Pos = 0 for timing master
Pos = 1 for first device in ring...

Logical Node Address:

Logical node addressing is used by all nodes to address a single node. Only the network master uses the node position address (administration only). 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:

$$\text{Rx/TxLog} = 0x0100 + \text{Pos}$$

So the system master device (containing the timing master) at position 0x00, will have the logical node address 0x0100, and a device at position 5 in the ring will have address 0x0105.

Another approach is to assign certain address ranges with respect to the functionality of devices. That means, for example, that the first video display module in a network gets address 0x0200, the second 0x0201, etc., while the first active amplifier gets address 0x0188.

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

For checking whether a logical node address is unique, the MOSTTransceiver provides a special procedure (SAI) which is described in section 3.1.5.2 on page 93.

The logical node address is stored in unbuffered RAM, so it is lost if the device loses power for some time. If the device stays powered, the logical node address is kept. After first power up, the logical node address is set to 0xFFFF (refer to section 3.2.3 on page 111).

Group Address:

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

$$\text{GroupAddress} = 0x0300 + \text{FBlockID}$$

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

Groups can be built dynamically by modifying group addresses.

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

Broadcast Address:

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

FUNCTIONS				
FktID	OPType	Sender	Receiver	Explanation
NodePositionAddress	Get	Controller	NetBlock	Requesting Node Position Address
	Status	NetBlock	Controller	Answer
NodeAddress	Get	Controller	NetBlock	Requesting Logical Node Address
	Status	NetBlock	Controller	Answer
GroupAddress	Get	Controller	NetBlock	Requesting Group Address
	Status	NetBlock	Controller	Answer
	Set	Controller	NetBlock	Setting Group Address

Table 3-12: Functions in NetBlock that handle addresses

3.3.2 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_{Transceiver}. Four priorities are defined that are assigned to different function blocks via the mXCMB Register (Default = 0x01; 0x00 = lowest Priority):

Priority	Used for function blocks that
0x00	are controlled
0x04	have administrative tasks
0x08	handle control events initiated by the end user
0x0C	reserved

Table 3-13: Priorities on the control channel

3.3.3 Low Level Retries

In case the sending of a control message is not successful, MOST_{Transceiver} can re-send the message automatically. Registers specify the number of retries and the delay between the retries. Typically, these values should not be changed, however they can be modified to fine tune the system.

3.3.4 High Level Retries

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

3.3.5 MOSTNetServices (Application Socket)

3.3.5.1 Basics for Automatic Adding of Physical Address

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

One way is when the application answers a request. In this case it already has the DeviceID of the receiving 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 receiver. In this case it sets the DeviceID to 0xFFFF. The ID is complemented by the NetServices and inserted into the MOST telegram as RxAdr.

3.3.5.2 De-Central Registry

The complementing of the DeviceID assumes that the assignment between a functional address (consisting of FBlockID.InstID and the logical node address) is known by the NetServices. If required, the NetServices build a De-central Registry, which contains the respective assignments:

Functional address (FBlockID.InstID)	Device containing the FBlock (logical node address = DeviceID)
AudioAmplifier.1	0x0105
AudioAmplifier.2	0x0103
AM/FMTuner.0	0x0107
AudioDiskPlayer.1	0x0107

Table 3-14: Example for a De-central Registry.

Only devices that control other devices need a De-central Registry. Devices that are controlled only, that is, those that receive requests and answer only those requests, for example, drives or tuners, get the DeviceID of the requesting node along with the request, and so do not need a De-central Registry.

To build a De-central Registry, i.e. for seeking a function block in the network, there are different possibilities. If there is a Central Registry, a request should be sent there, since this approach saves resources. See the next section for more explanation.

3.3.5.3 Central Registry

In larger networks it might be useful to build a Central Registry. The network master generates it during initialization of the network. It is kept in buffered RAM (i.e. it is deleted if the device is removed from power) and contains the logical node address for each node, and the respective function blocks:

Rx/TxLog	Rx/TxPos	FBlockID	InstID
0x0100	0	AudioDiskPlayer	1
		NetworkMaster	0
		ConnectionMaster	0
0x0101	1	AudioDiskPlayer	2
0x0102	2	AM/FMTuner	0
		AudioTapeRecorder	0
0x0103	3	AudioAmplifier	2
etc.			
MaxNode	MaxNode	ManMachineInterface	0

Table 3-15: Central Registry

The Central Registry checks the system configuration, and finds communication partners, or their physical addresses.

The Central Registry is saved on SystemShutDown. On a restart of the system, the network master asks all nodes for the information shown above. It then receives the logical node address automatically.

```
NetworkMaster -> Device: NetBlock.Pos.FBlockIDs.Get
Device -> NetworkMaster: NetBlock.Pos.FBlockIDs.Status (FBlockID.InstID,
FBlockID.InstID...)
```

If there is a contradiction between an entry in the Central Registry and the received data, the network master rejects the stored Central Registry and broadcasts:

```
NetworkMaster -> all: NetworkMaster.0.Configuration.Status (NotOK)
```

This forces all devices to reject their De-central Registries, and to re-initialize their logical node addresses (based on node position). The De-central Registries are re-built on demand, i.e., not directly after Configuration.Status (NotOK). This means an entry of the De-central Registry is built if the application in a device desires to send a protocol without knowing the physical address of the receiving node.

If there is no contradiction between entries in the Central Registry and the received data after all nodes have been requested, the network master broadcasts:

`NetworkMaster -> all: NetworkMaster.0.Configuration.Status (OK)`

Based on that, all other devices restore their De-central Registry based on the stored copy.

FUNCTIONS				
FktID	OPType	Sender	Receiver	Explanation
FBlockIDs	Get	Network Master	NetBlock Slave	Request of all function blocks contained in a device
	Status	NetBlock Slave	NetBlock Network Master	Answer
Configuration	Status	Network Master	NetServices of all Slaves	Event, by which the Network Master tells all slaves whether the system configuration is identical to the one of the last system run

Table 3-16: Commands in conjunction with the Central Registry.

If available, the Central Registry must be used if the NetServices of a device seek a physical address. With the help of the telegram Configuration.Status, the slaves recognize the logical address (Rx/TxLog) of the network master containing the Central Registry. They ask the network master for the physical address of a desired function block. The network master takes the respective logical node address (Rx/TxLog) from the Central Registry and sends it to the slave. The slave's NetServices ask:

`NetworkMaster.0.CentralRegistry.Get (FBlockID.InstID)`

The network master answers:

`NetworkMaster.0.CentralRegistry.Status (Rx/TxLog.FBlockID.InstID)`

If there is no matching entry in the Central Registry, the network master answers:

`NetworkMaster.0.CentralRegistry.Error (ErrorCode, ErrorInfo)`

with ErrorCode = 0x07 (Parameter not available) and returning of the parameter number (1 in this example), and the parameter which is not available (FBlockID.(InstID) in this example) as ErrorInfo.

It is possible to request the entire Central Registry:

`NetworkMaster.0.CentralRegistry.Get ()`

The network master replies with all entries of the Central Registry:

`NetworkMaster.0.CentralRegistry.Status (Rx/TxLog.FBlockID.InstID,
Rx/TxLog.FBlockID.InstID...)`

FUNCTIONS				
FktID	OPType	Sender	Receiver	Explanation
CentralRegistry	Get	NetServices Slave	Network Master	Requesting a physical address
	Status	Network Master	NetServices of requesting Slave	

Table 3-17: Functions for building a De-central Registry if a Central Registry is available.

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

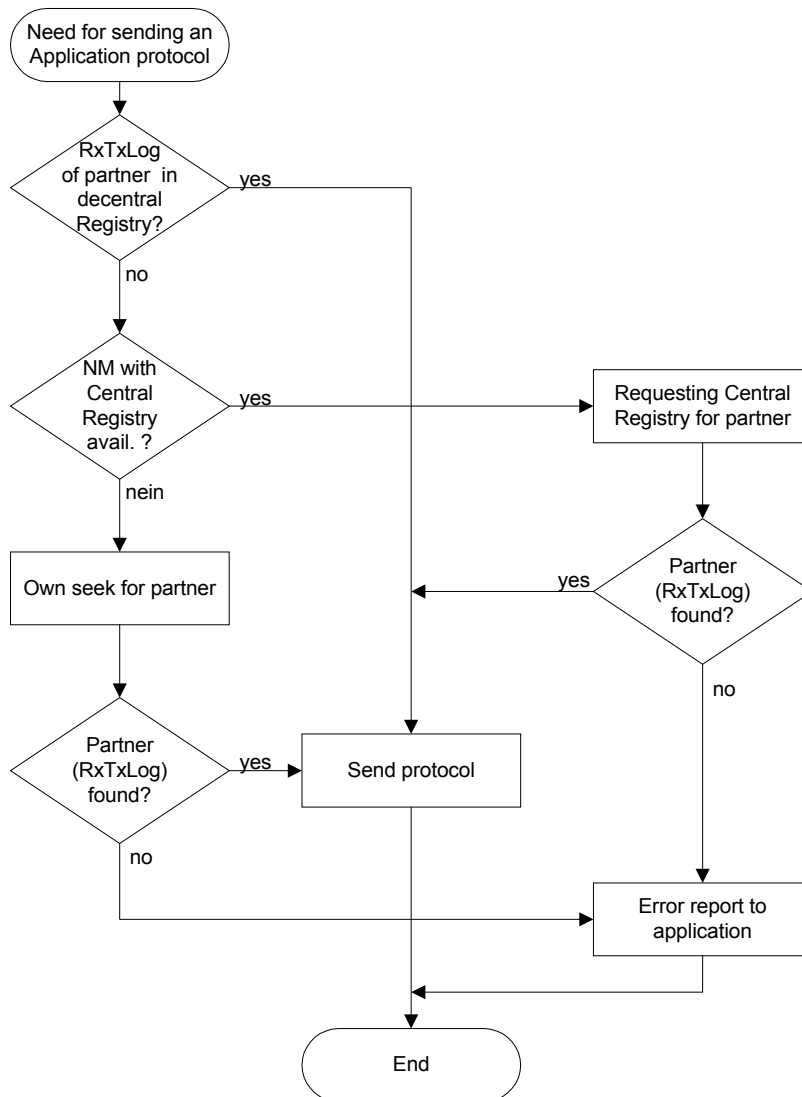


Figure 3-20: Seeking the logical address of a communication partner

3.3.6 Handling Overload in a Message Sink

The MOST_{Transceiver} informs the sender's NetServices 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 NetServices pass the NAK error message through to the application, which has to decide what needs to be done (retry, reject, etc.). 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, a mechanism can be implemented on demand that adapts the telegram transfer rate to the speed of the data sink. A simple mechanism might look like this:

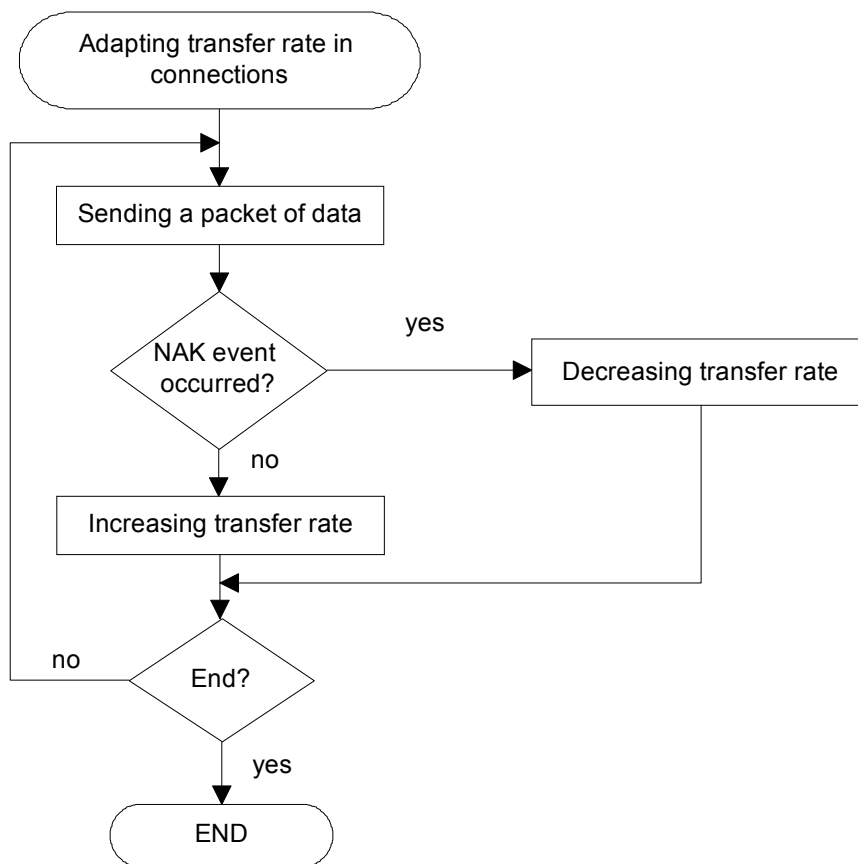


Figure 3-21: Possible mechanism to adapt transfer rates to the speed of a data sink during segmented transfer.

3.3.7 MOSTNetServices (Basic Layer)

3.3.7.1 Control Message Service

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

Data area of MOST Transceiver = 17 Byte



The lowest layer of the NetServices provide a mechanism which is called control message service (CMS). It handles the setting and reading of the registers of the MOSTTransceiver.

3.3.7.2 Application Message Service (AMS) And Application Protocols

MOSTNetServices provide three different types of transmissions via the control channel. Two of them are mandatory for each device:

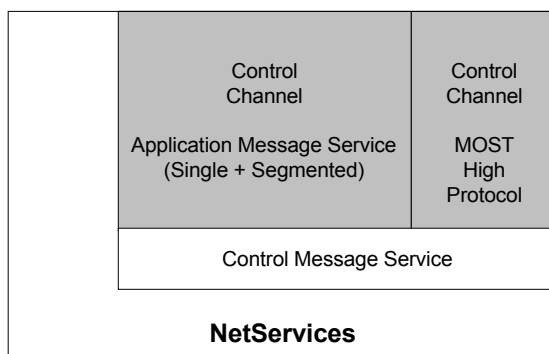


Figure 3-22: NetServices: Services for control channel

- **Single Transfer:** Data packets up to twelve 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.

Single as well as segmented transfers are based on the application message service (AMS), which is mandatory for all NetServices. 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 MOSTNetwork. 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. For more detailed information about MOST High Protocol, please refer to [3].

As already described in section 2.3.2 on page 29, 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 Transceiver = 17 Byte

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

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

TelID: Identification of kind of telegram

Meaning	TelID	Data 0 = MsgCnt
Single Transfer	0	Data 0
1 st telegram Segmented Transfer	1	0x00
2 nd telegram Segmented Transfer	2	0x01
..	2	..
..	2	0xFF
..	2	0x00
..	2	..
(n-1). Telegram Segmented Transfer	2	0x(n-1)
Last telegram Segmented Transfer	3	0xn
MOST High Protocol User data	8	
MOST High Protocol Control data	9	

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

Data 0-Data 11: Data bytes

3.3.8 Direct Access to OS8104

3.3.8.1 Sending Messages

In order to transmit a message, the “Message Transmit Section” must be loaded by an external micro controller. The address of the receiving node (or the group of nodes) that should receive the message, the message type, the priority, and the message itself must be written here. The message type specifies whether the message is a standard control command, a remote command, or a resource command. Setting the retry values should be done by external network management, and should not be changed for each message separately.

After loading the message, only the “Start of Transmission Bit (STX)” in the message control register (bMSGC) must be set. After transmission the chip generates an interrupt (only if enabled), which indicates that the status of transmission (successful or not) is available in the respective registers.

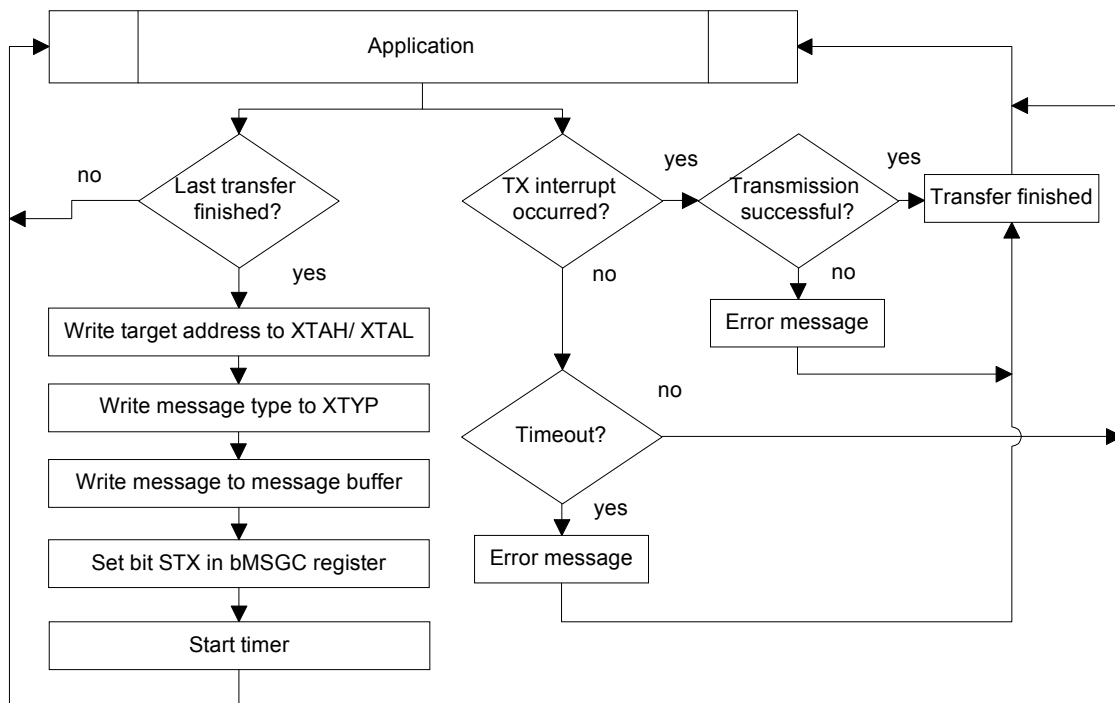


Figure 3-23: Sending a message on the control channel (Low level).

For more detailed information, please refer to [7].

3.3.8.2 Receiving Messages

An incoming message is recognized by checking the “Message Received Bit” in the message status register. This can be done either by polling, or via interrupt.

If an incoming message was recognized, the “Message Receive Section” must be read by the external micro controller. This section contains the received message type, the message itself, and the sender’s address.

After reading the message, the receive buffer must be unlocked with the help of the “Receive Buffer Enable Bit” (RBE) in the message control register. This mechanism avoids the loss of data for incoming messages at a high frequency.

For more detailed information, please refer to [7].

3.3.8.3 Acknowledgement and Data Security

The transmission of control data is secured with the help of a CRC. If CRC check fails on a received message, this is reported to the sending transceiver by an error mechanism. This mechanism repeats the sending of the failed message until either the transmission is successful or the maximum number of retries (which can be set by the application) has been reached.

If the maximum number of retries is reached, this is reported to the application by an error message. Depending on the definition of the external network management, the message may be resent by the application, eventually with a changed maximum number of retries. Re-initialization of the device that does not respond is another possibility.

Another mechanism for securing transmission of control data is a handshake between the external micro controller and the MOST_{Transceiver} on incoming messages. Another message can be received only if an incoming message was read, and the receive buffer of the transceiver is unlocked by the micro controller. If another node tries to send a message to a node with a locked receive buffer, this message is rejected by the receiving transceiver. The same retry mechanism is activated as used on CRC errors. How the external network management reacts to an error of this kind must be defined for this case.

For more detailed information, please refer to [7].

3.3.9 Remote Control

Remote control is a special functionality of the MOSTTransceiver. It does not influence the transmit buffer, nor the receive buffer of the remote controlled device, that is, its application stays untouched by this operation. For more detailed information, please refer to [7].

3.3.9.1 Remote Read Message

In order for another node to read the memory area of a MOSTTransceiver, a special kind of message must be used. This message is different in terms of the message type written into the XYTP register (0xC1). Here, the value 0x01 (Remote read) must be stored. XMIT control data byte 1 (located at 0xC5) contains the address from which data should be read. Beginning at this address, 8 bytes will always be read. After having sent the message, a transmit interrupt is released if enabled. If the transmission is marked as successful, the result can be read from the XMIT control data bytes 3 through 10 (0xC7 .. 0xCE).

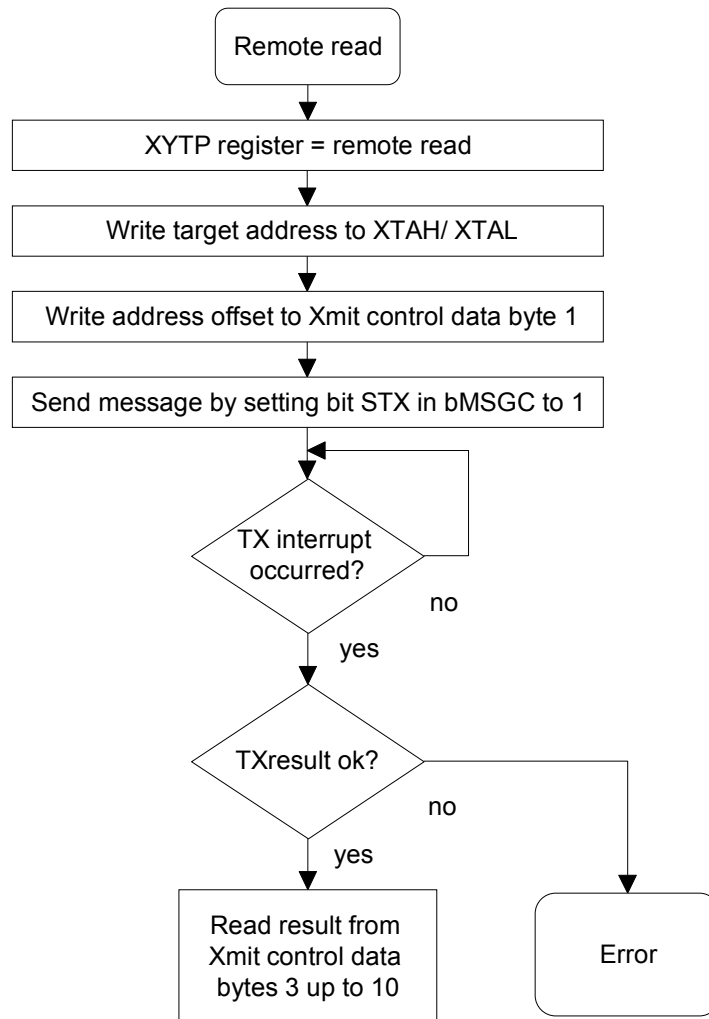


Figure 3-24: Remote read.

3.3.9.2 Remote Write Message

The remote controlled writing to registers of other MOST_{Transceiver} chips is done in principle as it is on the remote read access. Compared with remote read, the telegram is extended by the number of bytes to write. This information is written to register XMIT control data byte 2 (0xC6). Message type (XTYP) Remote Write (0x02) must be used. The data must be written to the XMIT control data bytes 3 through 10 (0xC7 .. 0xCE). Up to eight data bytes can be written per remote write access.

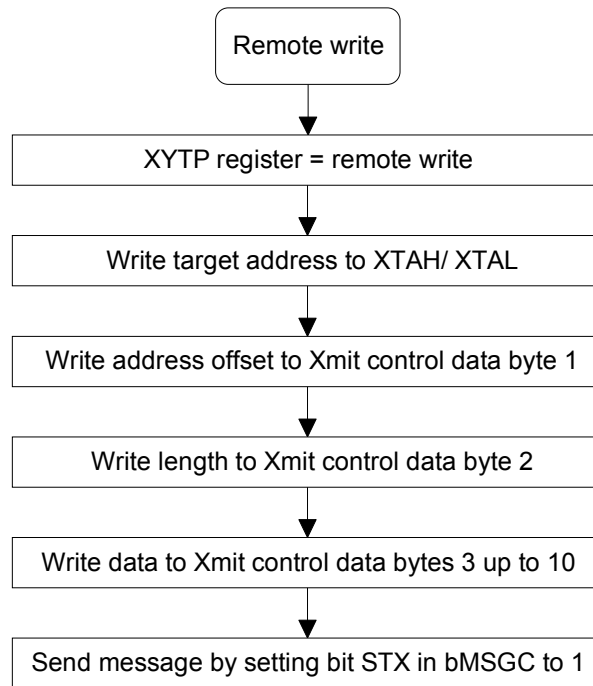


Figure 3-25: Remote write.

3.4 Handling Synchronous Data

3.4.1 MOSTNetServices (Application Socket)

The MOSTTransceiver already provides convenient mechanisms for administrating synchronous channels. In addition to that, the NetServices have routines that use those mechanisms to provide simple and flexible access to the synchronous resources of the network.

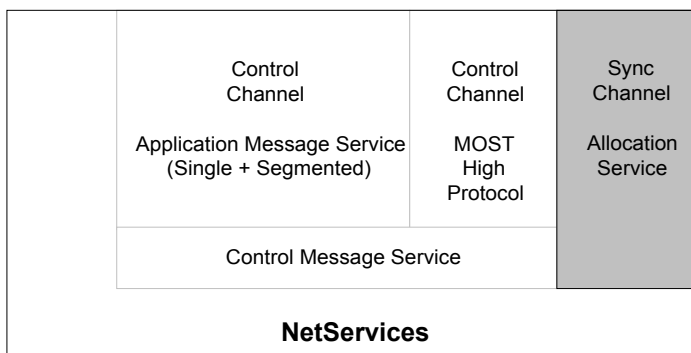


Figure 3-26: NetServices for the synchronous channel

On the network, 60 bytes for synchronous and asynchronous transport are available per frame. A certain number of these channels can be used for synchronous data transfer. Here, several channels can be clustered to a synchronous connection for an application. Every connection that does not use more than 8 channels gets a single handle (Connection Label), which is identical to the lowest number of used channels.

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 MOSTTransceiver in several different modes. Connecting the source ports with the channels is controlled via the Routing Engine (RE).

The following routines are implemented in the NetServices:

- **SyncAlloc (Source):**
Allocating channels for synchronous data transfer and routing the data from the source data port of the MOSTTransceiver to these channels via RE.
- **SyncDealloc (Source):**
Restoring RE, i.e., direct routing of the formerly-occupied channels from input to the output of the MOSTTransceiver, and then de-allocation of the channels.
- **SyncOutConnect (Sink):**
Routing of the channels specified in the function call from the input of the MOSTTransceiver to the source ports of a sink (also specified in the function call).
- **SyncOutDisconnect (Sink):**
Restoring RE in a sink, i.e., removing the connection between the synchronous channels on the network and the source data ports of the MOSTTransceiver.

- **SyncFindChannels:**
Seeking channels belonging to a certain handle.
- **MostSetBoundary (Timing Master):**
Modifying the boundary between synchronous and asynchronous data transfer.

3.4.1.1 Basic Functions on Application Level

On the application level, different basic functions in sources and sinks are realized, which serve the administration of synchronous connections. They themselves access the routines of the NetServices. The synchronous data transfer does not need to be modeled functionally.

3.4.1.1.1 NetBlock

NetBlock contains the function **SourceHandles**, by which a controller can learn which function block in the slave holds a connection. Therefore it asks:

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

and gets the function blocks that work with the respective handle in the answer. There can be multiple function blocks in a device:

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

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

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

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

FUNCTIONS				
FktID	OPType	Sender	Receiver	Explanation
SourceHandles	Get	Controller	NetBlock	Requesting function blocks working with a certain handle.
	Status	NetBlock	Controller	Answer

Table 3-18: Functions in NetBlock in conjunction with the administration of synchronous resources.

3.4.1.1.2 Function Block

Also in a function block that contains an application, different functions for administering synchronous connections are implemented. By using function SyncDataInfo it can be requested, for how many connections the function block may serve as source (parameter SourceCount, 8 bits), or as sink (parameter SinkCount, 8 bits):

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

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

Functions				
FktID	OPType	Sender	Receiver	Description
SyncDataInfo	Get	Controller	Function block	Request of the number of synchronous sources and sinks within a function block.
	Status			Answer

Table 3-19: Common functions in a function block for administering synchronous resources

3.4.1.1.2.1 Synchronous Source

Property **SourceInfo** contains more detailed information about the kind of synchronous source data. On a request with the SourceNr (8 bits starting at 0x01; if a function block has only a single source, it always has 0x01):

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

one receives

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

The parameter **DataType** describes the kind of synchronous data stream that is sent by the source. Depending on DataType, a DataDescription may follow. The following synchronous data types are defined at the moment:

Data type 0x00 Audio:

Here, the additional parameters Resolution, AudioChannels, Delay and Handle/Channels are specified.

`DataDescription = Resolution, AudioChannels, SrcDelay, Channels`

Parameter **Resolution** (1 Byte) specifies the resolution of audio samples in bytes. Parameter **AudioChannels** (1 Byte) specifies the number of audio channels, e.g., 1 for mono, 2 for stereo etc. Parameter **SrcDelay** (1 Byte) specifies the delay of the synchronous data with respect to the timing master. It is equal to the contents of the node delay register (NDR) of the MOSTTransceiver. In the last parameter, the single **Channels** (1 byte per channel) are listed. The first channel corresponds to the handle. If the source has not allocated channels at the moment, it returns 0xFF.

MOSTTransceiver is able to receive many different audio formats and convert them to its raw data format, or to generate many different audio formats from the transported raw data. For audio transmissions, the following minimum appointments are valid:

- Audio-NF will be transported CD-DA compatible (Compact Disk Digital Audio)
- The sequence of channels is: Front left, front right, rear left, rear right. The most significant byte is transmitted first.

Examples:

16 Bit Stereo: Resolution = 0x02, Channels = 0x02,
Sequence on the bus : MSB left, LSB left, MSB right, LSB right.

24 Bit Stereo: Resolution = 0x03, Channels = 0x02,
Sequence on the bus : MSB left, central byte left, LSB left, MSB right, central byte right, LSB right.

If the property SourceInfo is not implemented by the source, data type audio is assumed by default.

Data type 0x01 CD ROM:

This data type describes CD-ROM raw data before being processed by a CD-ROM decoder. This data might be of type audio, or CD-I, or Video-CD respectively.

`DataDescription = Blockwidth, Channels`

For data type CD-ROM, parameter "Blockwidth" is transmitted. It specifies the number of transmitted bytes per MOST frame.

Examples:

Single Speed CD : Blockwidth = 0x04
Double Speed CD : Blockwidth = 0x08

Per Default, Blockwidth = 0x04 will be assumed.

In property **SourceName**, a name for the synchronous source data can be requested:

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

As an answer, a string is returned that contains the required name:

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

The source is induced to allocate synchronous channels by function **Allocate** :

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

The function block accesses the respective routines of the NetServices (SyncAlloc as instruction, and SyncAllocComplete as answer) and reports the following result on success:

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

Error handling:

If the allocation was not successful due to a lack of enough free channels, an error code is generated with error code "Function specific" and as Error Info the SourceNr and the required channels containing 0xFF (not allocated channels):

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

DeAllocate induces the source to de-allocate allocated synchronous channels:

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

The function block accesses the respective routines of the NetServices (SyncDealloc as instruction and SyncDeallocComplete as answer) and sends the result:

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

FUNCTIONS				
FktID	OPType	Sender	Receiver	Explanation
SourceInfo	Get	Controller	Application of the source	Request of information about the kind of synchronous source data
	Status	Application of the source	Controller	Answer with parameters depending on type
SourceName	Get	Controller	Application of the source	Requesting the name of the synchronous source data
	Status	Application of the source	Controller	Answer as string
Allocate	StartResult	Controller	Application of the source	Demand for allocation
	Result	Application of the source	Controller	Reply with result of allocation
DeAllocate	StartResult	Controller	Application of the source	Demand for de-allocation
	Result	Application of the source	Controller	Reply with result of de-allocation

Table 3-20: Functions in a function block with a synchronous source, in conjunction with administering synchronous resources.

3.4.1.1.2.2 Synchronous Sink

In a function block that is used as a sink for synchronous data, analog functions like those for a source are implemented. Error handling is also done in an analogous way. Via function **SinkInfo**, information about which kind of data the sink can handle is stored. Here the SinkNr specifies the desired sink in the function block (8 bits starting at 0x01; if a function block has only a single sink, it always has SinkNr 0x01):

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

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

The parameters are identical to those described above, except that SrcDelay must be replaced by SinkDelay.

SinkName is available to request a name for the synchronous data:

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

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

Function **Connect** induces the sink to connect to certain channels:

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

It accesses the respective routine of the NetServices (SyncOutConnect). SrcDelay is the delay that is passed to the sink to provide the possibility of delay compensation. The sink returns as result:

```
Slave -> Controller: FBlockID.InstID.Connect.Result (SinkNr),
```

Function **DisConnect** induces the sink to remove a certain connection:

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

It accesses the respective routine of the NetServices (SyncOutDisconnect) and returns as result:

```
Slave -> Controller: FBlockID.InstID.DisConnect.Result (SinkNr),
```

In many cases, the output of synchronous data on a sink must be stopped. For this, the function **Mute** is used:

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

with Status = [On/Off].

In smaller systems it might be useful to use a generic peer-to-peer approach for establishing connections. Therefore the sink is provided with function **ConnectTo**:

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

With this method, the sink is induced to connect to a certain source. The sink uses SourceInfo to check whether the source sends a signal which it can handle. Then source and sink connect without external influence. On success the sink reports:

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

FUNCTIONS				
FktID	OPType	Sender	Receiver	Explanation
SinkInfo	Get	Controller	Application of the sink	Request of information with respect to the kind of the synchronous data the sink can handle
	Status	Application of the sink	Controller	Answer with parameters depending on type
SinkName	Get	Controller	Application of the sink	Requesting the name of the synchronous data the sink can handle
	Status	Application of the sink	Controller	Answer as string
Connect	StartResult	Controller	Application of the sink	Demand for connecting to certain channels
	Result	Application of the sink	Controller	Reply with result
DisConnect	StartResult	Controller	Application of the sink	Demand for removing a connection
	Result	Application of the sink	Controller	Reply with result
Mute	Set	Controller	Application of the sink	Starting/ stopping the output of the synchronous data
ConnectTo	StartResult	Controller	Sink	Demand for building a connection to a certain source
	Result	Sink	Controller	Reply with result

Table 3-21: Functions in a function block with a synchronous sink in conjunction with the administration of synchronous resources.

The basic functions described to this point provide the detailed requesting of sources and sinks for information about source data that can be handled. This is the basis for a flexible, self-configuring system. It is possible to implement a peer-to-peer approach, where sink and source act more independently from controlling devices:

Only source and sink must have information about the handling of their own data types. The MMI or a higher controller does not need to handle some types of information, e.g., SourceInfo. A data sink checks whether the data type of the source matches its own data type before building the connection. This means that data sinks need to know only the parameters of their own data types, which allows the data type definition to be extended easily if required.

3.4.2 MOSTNetServices (Basic Layer)

When using the basic layer of the MOSTNetServices, the allocation and de-allocation of channels for transporting synchronous source data is possible, but extra functions are required to inform the NetBlock of allocation results.

3.4.3 Direct Access to OS8104

3.4.3.1 Serial Interface

Up to 4 source data ports are available for input and output of synchronous source data in the MOSTTransceiver. Each source data port can handle up to 64 bits per frame. To achieve higher data rates, there is the possibility of cascading several source data ports. Nevertheless, the number of source data ports is decreased by this.

Every port has the ability to handle different formats, which can be specified by control registers. For more detailed information please refer to [7].

3.4.3.2 Parallel Interface

In addition to the ability to input or output source data in serial mode, the MOSTTransceiver also has a parallel interface. Asynchronous or control data can also be input or output via this parallel interface.

A FIFO method provides buffering for the purpose of external synchronization. Data flow is handled via the respective control pins of the chip.

3.4.3.3 Compensating Network Delay

Every active node in the ring (source data bypass inactive), generates 2 samples delay for the source data (caused by internal processing). Especially in top HIFI applications, this is an unpleasant effect. The MOST system therefore provides mechanisms that allow compensation for this delay. Every chip is provided with the information about the general delay of the entire system $\Delta T_{\text{Network}}$, and the delay up

to its own node ΔT_{Node} with respect to the timing master. In addition to that, the delay of the active source device ΔT_{Source} must be made available by control messages.

Based on that information, the delay ΔT_{comp} , which must be compensated for, can be calculated with the help of the formula below:

$$\begin{aligned} \Delta T_{comp} &= \Delta T_{Source} - \Delta T_{Node} - 2 \quad \text{[Samples]} \quad \text{for } \Delta T_{Source} < \Delta T_{Node} \\ \Delta T_{comp} &= \Delta T_{Network} - \Delta T_{Source} + \Delta T_{Node} - 2 \quad \text{[Samples]} \quad \text{for } \Delta T_{Source} > \Delta T_{Node} \end{aligned}$$

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

3.5 Handling Asynchronous (Packet) Data

3.5.1 Direct Access to OS8104

A data packet that can be sent in the asynchronous area consists of 48 bytes of data and a 16 bit receiver address (logical node address only):

Data area MOST Transceiver = 48 Byte

16 Bit Rx/TxLog	
--------------------	--

The data field is significantly longer than that of the control channel. Unlike the control channel, no ACK/NAK mechanism or low level retries are implemented, since they are not necessary for most of the applications. Nevertheless, the telegram is checked. A transport protection must be implemented on a higher level.

3.5.1.1 Priorities

In every node (MOSTTransceiver) the priority for packet data transfer can be specified. It is based on the control of access authorization of a node on free frames in the data stream on the bus:

Range: 0x1..0x7
Default: 0x1 (Highest Priority)
Register: bPPI

At first, only priority 0x1 is used.

3.5.2 MOSTNetServices

The NetServices have a mechanism, called the Asynchronous Data Transmission Service, by which the data packets described above can be sent and received. It handles the respective registers in the MOSTTransceiver.

3.5.2.1 Securing data

For stream data of high bandwidth, e.g., graphical applications, it is not useful to implement mechanisms for secure data transmission. On one hand the data security (bit error rate) of MOST is approximately 10^{-12} , on the other hand every securing mechanism must be checked by a μ Controller, which becomes more and more difficult, as the bandwidth is increased. In case of errors, transmissions would be repeated, which would cause delays that might be unwanted. It must be decided for each application, whether mechanisms for secure data transmission would be useful, and if so, which implementation to use.

For certain applications which transmit in the asynchronous area at a lower bandwidth, it might be useful to implement securing mechanisms. The telegram structure quite alike that of the control channel could be used by setting TelID to 4 bits and TelLen to 12 bits.

Data area MOST Transceiver = 48 Byte (Data Link Layer 48 Bytes Mode)

16 Bit	8 Bit	8 Bit	12 Bit	4 Bit	4 Bit	12 Bit	8 Bit	8 Bit	...	8 Bit
DeviceID	FBlock ID	Inst. ID	Fkt ID	OP Type	Tel ID	Tel Len	Data 0	Data 1	...	Data 41

Data area MOST Transceiver = 1014 Byte (Alternative Data Link Layer)

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

TelID: Identification of kind of telegram

Meaning	TelID
MOST High Protocol User data	8
MOST High Protocol Control data	9
Reserved for Ethernet frames	A

TelLen: = up to 1008 (42)
Specifies the length of the data field, i.e. ,the number of bytes after TelLen

Data X: Data bytes

For securing data, MOST High Protocol is used here. For more detailed information, please refer to [3].

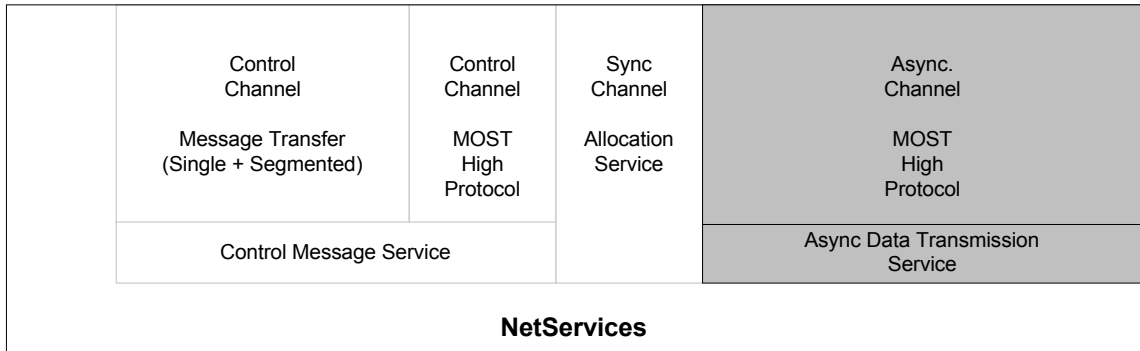


Figure 3-27: NetServices: Services for the asynchronous channel.

3.6 Controlling Synchronous/Asynchronous Bandwidth

When administrating the boundary between synchronous and asynchronous data area, two contrary requirements must be taken into consideration. On one hand, there should be as much bandwidth as possible for asynchronous data transfer, so it is not reserved for unused synchronous channels. On the other hand, the boundary should be changed only in rare cases, since all synchronous connections must be re-allocated after the boundary was changed.

Even with the most adverse usage of a fully equipped vehicle, the entire available bandwidth for synchronous transfer is seldom used. Generally, less bandwidth is required for synchronous transfer.

System initialization adjusts the boundary to the center of the bandwidth. During runtime it is shifted only "to the right", that is, in the direction of an extension of the synchronous area, up to a limit, which will reserve, e.g., one quadlet for asynchronous data transfer. There is no shifting to the left. The boundary is returned to its default value only after a re-initialization of the system.

The changing of the boundary between synchronous and asynchronous area is done physically by the timing master, located in the system master device. Timing master functionality is provided by a MOSTTransceiver, so the network master therefore provides function **Boundary**. It is controlled by the Connection Master:

```
??? -> SystemmasterDevice: NetworkMaster.0.Boundary.Set (BoundaryDescriptor)
```

The parameter BoundaryDescriptor corresponds to the one used in the MOSTTransceiver, and can be written directly into that register by the network master.

The current status of the boundary can be requested by:

```
??? -> SystemmasterDevice: NetworkMaster.0.Boundary.Get
```

The answer is:

```
SystemmasterDevice -> ???: NetworkMaster.0.Boundary.Status  
(BoundaryDescriptor)
```

FUNCTIONS				
FktID	OPType	Sender	Receiver	Explanation
Boundary	Set	Connection Master	Network Master	Setting of boundary between synchronous and asynchronous area
	Get	Controller	Network Master	Request for status of boundary between synchronous and asynchronous area
	Status	Network Master	Controller	Answer on request

Table 3-22: Functions for administration of boundary between synchronous and asynchronous area in network master.

3.7 Connections

3.7.1 Synchronous Connections

3.7.1.1 Administering (Connection Master)

For managing synchronous connections in complex networks it might be useful to introduce a higher control instance on the application level. It will be implemented in function block ConnectionMaster. Since the connection master must verify that a requested connection does not already exist, all requests for establishing connections must be directed to the connection master. For building a point-to-point connection, it provides the method **BuildSyncConnection**:

```
Controller -> ??? : ConnectionMaster.0.BuildSyncConnection.StartAck
                    (SenderHandle, Source, Sink)
```

The question marks are used in this example to indicate any device, since it is not relevant which device the connection master is implemented in. Source stands for the source of a connection with:
Source = FBlockID.InstID.SourceNr.

Sink is the sink of the connection with:
Sink = FBlockID.InstID.SinkNr.

After a successful connection, the connection master returns:

```
??? -> Controller : ConnectionMaster.0.BuildSyncConnection.ResultAck
                    (SenderHandle, Source, Sink)
```

Error handling:

If the connection fails, the connection master answers with OPType "ErrorAck" (0x9) and the ErrorCode "ProcessingError" (0x42), and returns the parameters Source and Sink:

```
??? -> Controller : ConnectionMaster.0.BuildSyncConnection.Error
                    (SenderHandle)
```

Removing a connection is done in an analogous way, with the help of the **RemoveSyncConnection** method.

The connection master generates an array of all existing connections including sources and sinks, where it adds more information. This array is accessible in function **SyncConnectionTable** :

```
??? -> Controller : ConnectionMaster.0.SyncConnectionTable.Get

Controller -> ??? : ConnectionMaster.0.SyncConnectionTable.Status
                    (Source, Sink, SrcDelay, Channels,
                     Source, Sink, SrcDelay, Channels,
                     Source, Sink, SrcDelay, Channels,...)
```

The parameters are the same as those described above. SyncConnectionTable can not be set directly. Building and removing connections is done only with BuildSyncConnection and RemoveSyncConnection.

After switching off the network (light off), the contents of SyncConnectionTable are deleted, leaving no synchronous connections in the system. They must be rebuilt by new requests of the initiator(s).

FUNCTIONS				
FktID	OPType	Sender	Receiver	Explanation
BuildSyncConnection	StartAck	Controller	Connection Master	Request for building connection
	ResultAck	Connection Master	Controller	Answer with result
SyncConnectionTable	Get	Controller	Connection Master	Request of that property, where the Connection Master stores all active point-to-point connections
	Status	Connection Master	Controller	Answer
RemoveSyncConnection	StartAck	Controller	Connection Master	Request for removing connection
	ResultAck	Connection Master	Controller	Answer with result

Table 3-23: Functions in connection master in conjunction with the administration of synchronous connections.

3.7.1.2 Establishing Synchronous Connections

In building a synchronous connection, the connection master induces the source to allocate the required channels. After a positive answer from the source about the allocation of channels, the connection master can pass the data for building the connection to the sink. This mechanism is explained in the following figure, and the text below.

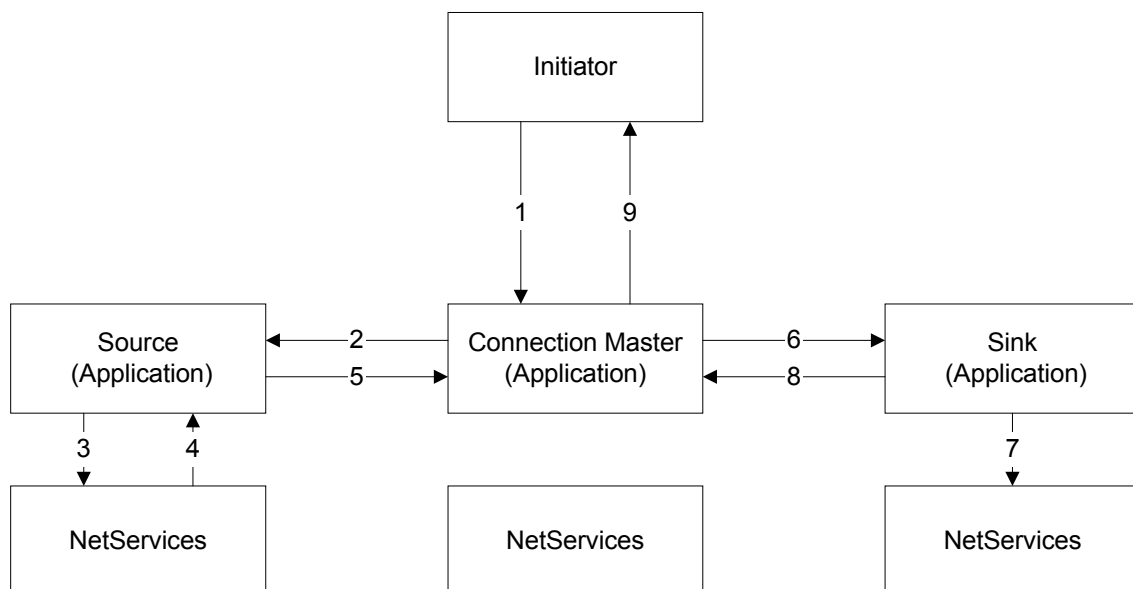


Figure 3-28: Flow of building a connection in synchronous area.

Step by step description:

1. Method *BuildSyncConnection* is started by an initiator (e.g., an MMI), for building a connection between a source and a sink.
2. If this source has not (yet) allocated a transfer channel, the connection master starts method *Allocate* for allocating a transfer channel, otherwise continue at step 6.
3. The application of the source calls routine *SyncAlloc* of the NetServices.
4. The NetServices returns the result to the application of the source by Callback Function *SyncAllocComplete*.

5. The application of the source processes the result:
 - ALLOC_GRANT: Enough free channels.
Reply to connection master as *Allocate.Result* with parameters SourceNr, SrcDelay and Channels.
 - ALLOC_BUSY: Timing master is busy on processing other allocation requests.
The source retries allocation (maximum 5 retries with 10ms delay).
 - ALLOC_DENY: Not enough channels.
Reply to connection master as *Allocate.Error* with parameters SourceNr and Channels, where Channels refers to the needed channels, the values however are set to 0xFF (e.g., 0x FF FF FF FF for 4 channels); continue at step 6.
6. If the result is *Allocate.Result*, the connection master starts method *Connect* of the sink, for routing data to the respective channel. For simplification, it passes the parameters Channels and SrcDelay of the source.
7. If the connection master receives the message *Allocate.Error* from the source, it can initiate moving the boundary between the synchronous and the asynchronous area. It writes the new value of the boundary descriptor to function Boundary of the function block NetworkMaster. Please note that all synchronous connections must be re-built after boundary was changed.
8. The application of the sink starts routine *SyncOutConnect* of the NetServices.
9. The sink reports the routing to the connection master as *Connect.Result*.
10. The connection master reports the result of establishing the connection to the initiator by using *BuildSyncConnection.ResultAck*. If the building of the connection was successful, the connection master 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. On failure, an error handling must be performed (e.g., displaying a message).

Some important connections are established once during system initialization, if the devices are available. The connecting is initiated by the connection master and the connections stay established constantly during operation.

Administration of connections is done only by the connection master. Other devices cannot detect whether connections are pre-installed or not. They behave as if there were no fixed connections, i.e. they apply to the connection master for establishment or termination of connections.

3.7.1.3 Removing Synchronous Connections

The initiator terminates a connection at the connection master. The connection master induces the sink to disconnect its routing connection to the transfer channel. After a positive answer, and if the channel is not in use by another sink, the connection master induces the source to de-allocate its channels. After that, the connection master reports the result of the termination to the initiator.

Step by step description:

1. An initiator starts method *RemoveSyncConnection* in the connection master, for terminating a connection between a sink and a source.
2. The connection master starts method *DisConnect* in the application of the sink, for disconnecting its routing connection to the transfer channels.
3. The application of the sink starts *SyncOutDisconnect* in its NetServices.
4. The application of the sink reports the result to the connection master by *DisConnect.Result*.
5. If no other sink occupies the transfer channel, the connection master starts method *DeAllocate* in the application of the source, for releasing the respective channels. Otherwise continue at step 10.
6. The application of the source starts routine *SyncDealloc* in its NetServices.
7. The source evaluates *SyncDeallocComplete* :
 - DEALLOC_GRANT: Successful de-allocation
Positive answer by *DeAllocate.Result*
 - DEALLOC_BUSY:
The timing master is busy handling other allocation/ de-allocation requests. The source tries another De-Allocation, up to a maximum of 5 retries (after 10 ms each).
8. The source's application answers the connection master with *DeAllocate.Result*
9. If the connection master has received a positive answer from the source, the respective connection is removed from its internal connection table.
10. By *RemoveSyncConnection.ResultAck*, the connection master sends an answer to the initiator, that contains the status of the requested termination of the connection.

3.7.1.4 Supervising Synchronous Connections

If a source that occupies synchronous channels has a defect, it closes its all bypass. By doing that, the source receives incorrect data. In this case, which can be recognized by a short unlock, and/or a NetworkChangeEvent, the reaction will be:

- Every slave in the network secures its output signals (analog or digital). A loudspeaker mutes, a display shows a default color, etc.

3.8 Timeouts

Name	Affects	Value	Meaning
Initialization			
t _{Config}	Each Device	100 ms	Maximum time between "light on" at the input of the FOT and the NetOn event.
t _{WakeUp}	Each Device	6 ms	Maximum time between "light on" at the input of the FOT and "light on" at the output of the FOT.
t _{PowerOn}	Each Device	10 ms	Time that might pass between the switching off of the reset at the MOST Transceiver and the moment when the transceiver generates a Power-On-Interrupt.
t _{Slave}	Timing Slave	2000 ms	Time that might pass (in a slave device) after initialization of the transceiver, until no more lock errors occur, and the bSBC register contains a value >5 (Ring closed).
t _{WaitNodes}	Each active Device	100 ms	Time that might pass, after "Light On" at the input of the FOT, until an active node has deactivated its bypass.
t _{Lock}	Each Device	100 ms	Time during which no Lock Errors must occur, for being able to declare the Lock as "stable".
t _{Master}	Timing Master	2000 ms	Time that might pass (in a master device) after initialization of the transceiver, until no more lock errors occur.
t _{CfgStatus}	Network Slave	3000 ms	Time a network slave will wait after NetOn, for receiving Configuration.Status from NetworkMaster.
t _{Answer}	Network Master	200 ms	Time the Network Master waits for an answer on the request for FBlockIDs.
t _{WaitAfterNCE}	Network Master	100 ms	Time the Network Master waits after a NetworkChangeEvent, until it retries to check system configuration.
Shut Down			
t _{ShutDown}	Each Device	15 ms	Time after which a device must have switched off its light at the output after a Shut Down event (Light off at the input of the FOT until light off at the output of the FOT).
t _{Suspend}	Power Master	2000 ms	Time the Power Master will wait for a ShutDown.Result (Suspended) after broadcast of ShutDown.Start, until it switches off light.
t _{ShutDownWait}	Power Master	1000 ms	Time the Power Master will wait after a ShutDown.Start (Execute), until it switches off light.
t _{RetryShutDown}	Power Master	10000 ms	Time the Power Master waits after ShutDown.Result (Suspend), until it retries to shut down the network by a new broadcast ShutDown.Start.

To be continued

continued

Name	Affects	Value	Meaning
Error Management			
$t_{Restart}$	Each Device	300 ms minimum	Time that must be waited for between switching off and switching on again of the network (Light off until Light on at the output of the FOT).
t_{Unlock}	Each Device	70ms	Unlock events with a length greater than t_{Unlock} are handled as errors in the network, otherwise unlocks are reported to the application for getting an eventual local reaction (e.g., Mute). The criterion is not only length, but also the frequency.
$t_{UnlockRecovery}$	Each Device	100ms	Time during which – after lock, no unlocks must occur. If there are no unlocks during that time, the lock can be regarded as stable. This is then reported to the application by the NetServices. The application then can restore the synchronous signals
Ring break diagnosis			
t_{Diag_Master}	Timing Master	2000 ms	Time a timing master waits for light at FOT input, during ring break diagnosis.
t_{Diag_Slave}	Slave Device	4000 ms	Time a slave device waits for light at FOT input, during ring break diagnosis.
t_{Diag_Lock}	Each Device	250 ms	On ring break diagnosis. Time during which there must not be any lock errors, for being able to declare the lock as "stable". Preliminary value.
Misc.			
$t_{OptPwrLow}$	Each Device	1000 ms	Time during which a device sends at reduced power, after a KWP2000 command. After expiration of this time, the device has to send at maximum optical power again.
$t_{PwrSwitchOffDelay}$	Each Device	5000 ms	Time after "light off" at the input, during which an application must stay active, since the light might be switched on again (If the switching off was caused by an error handling by network master).
$t_{MsgResponse}$	Each Device	15 ms	Maximum time after which a device must have read a control message from the RX buffer of the MOSTTransceiver. This determines the frequency by which the NetServices must be called.

Table 3-24: Timeouts

4 Hardware Section

4.1 Basic HW Concept

The fundamental hardware structure of a MOSTDevice is displayed in the block diagram below. There are blocks that are not mandatory, since, e.g., a simple MOSTDevice does not always need a micro controller (active speaker). Areas that are not mandatory are displayed in gray. A MOSTDevice consists of:

- Optical interface area
- MOST function area
- μ C area
- Application area
- Power supply area

A MOSTNetwork is awakened optically. All MOSTDevices are connected to a continuous power supply. They activate a sleep mode if required. If a device is in sleep mode, the power consumption should be reduced as far as possible (current < 100 μ A). For this reason, unused areas must be separated from the power supply. Only the sections that are absolutely necessary will stay powered. It must be taken into consideration that no parasitic currents will flow via signal lines between inactive and active sections.

The individual areas are explained below.

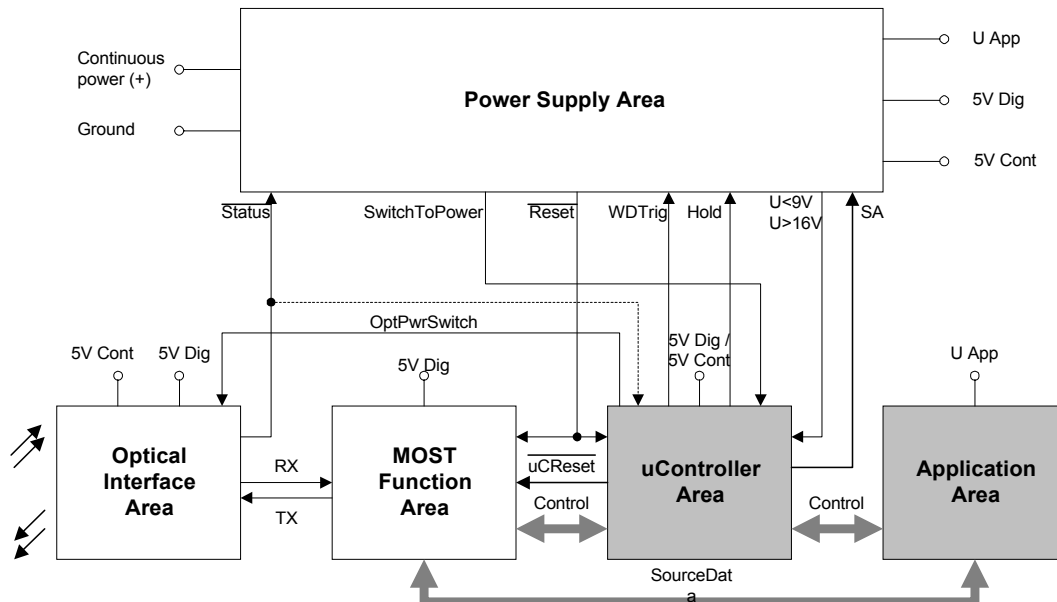


Figure 4-1: Structure of a MOSTDevice; The different functional areas and their interfaces.

4.2 Optical Interface Area

4.2.1 Overview

The optical interface area consists of an optical receiver and an optical transmitter. Both communicate with the MOST_{Transceiver} via a single data line (RX and TX). The receiver is connected to continuous power, unlike the transmitter.

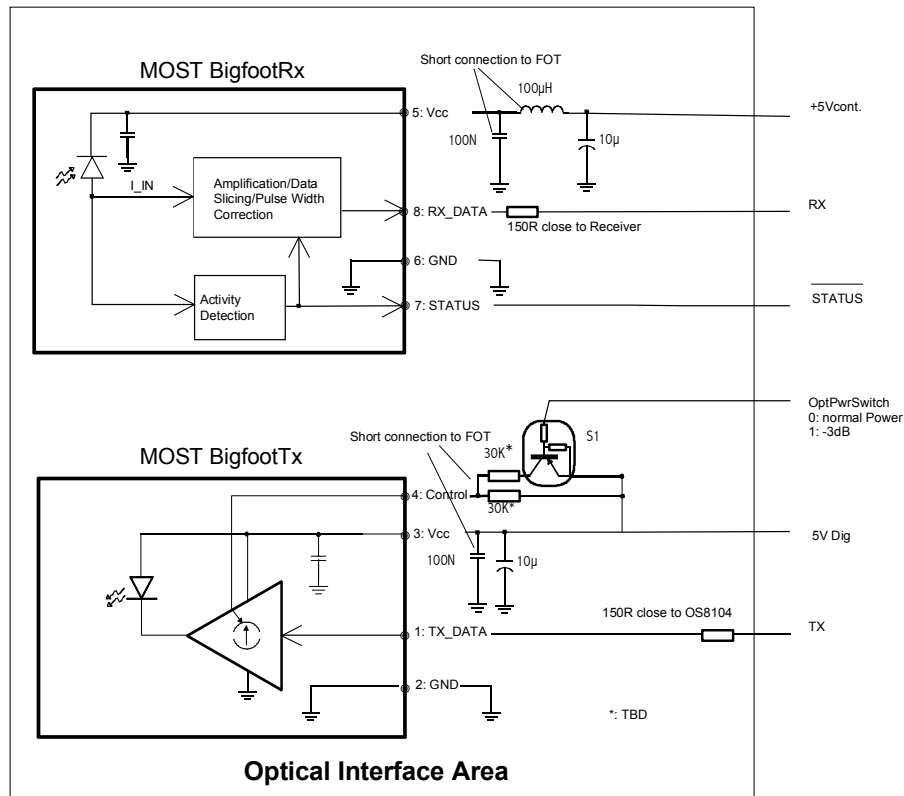


Figure 4-2: Optical interface area.

The receiver contains an ActivityDetection logic that is supplied with continuous power via the micro power regulator (5V cont.), and that consumes less than 20µA. As soon as the ActivityDetection logic recognizes modulated light, signal status is switched to low, and the received data stream is switched to the output.

Signal **Status** is connected to the power supply area, and therefore the power to the transceiver, and eventually to other areas, is switched on.

If no light is received, the receiver is switched off, except for the wake-up logic. Signal **Status** is high then.

It is possible to influence the driver current of the transmitter by a resistor between 5Vdig and input Control. A second resistor that has the same value, can be connected in parallel to the first resistor by using switch 1. When doing that, the optical output power is increased by approx. 3dB. Switch 1 is controlled by signal **OptPwrSwitch**, which is driven by the μ C area. The μ C only activates **OptPwrSwitch**, if it has received the respective KWP2000 command. After $t_{\text{OptPwrLow}}$, the μ C has to deactivate OptPwrSwitch independently.

In normal operation mode the switch is closed, so the transmitter runs at maximum optical power output.

This circuit provides diagnosis. If the optical power between two devices is reduced by 3dB and the system still works correctly, it can be assumed that there is a reserve of at minimum 3dB with respect to the optical power budget.

Please note:

By an appropriate arrangement (e.g. Pull down resistor, or inverter connected to OptPwrSwitch) it must be made sure, that S1 is closed in case of an inactive μ C.

For the lock behavior in a MOST network there are two important influencing variables:

- Optical Power Budget
- Phase jitter

In opposite to the power budget, the phase jitter may be accumulated when passing several nodes. An important influence variable for the phase jitter is the design of the optical interface area. Here interference's on highly resistive data wiring and crosstalk may occur.

For avoiding that, in the data lines to and from the MOST Transceiver a resistor of 100Ohms up to 150Ohms must be inserted. The resistors must be placed as close to the feeding output as possible. In addition to that, optical receiver and transmitter must be placed closely to the MOST Transceiver. The maximum length of data lines to Rx and from Tx must be less than 1.5cm.

Another important factor is the layout of the PCB. Below all data lines, transmitter and receiver a HF ground plane should be placed. Ideally, Rx and TX line are placed as far apart as possible, separated by a piece of ground area. The shielding box of the optical header must be connected well to the ground plane (by soldering). In addition to that, the power supply of the optical interface area must be buffered and blocked carefully. Therefore the bypass capacitors must be placed as close to the transmitter and receiver as possible. 100nF (Ceramic type) must be placed between every VCC and GND. Another important point is, that the bypass capacitors of transmitter and receiver must be located between the transmitter/ receiver and that point, where the two ground planes of receiver and transmitter are connected together.

Please note

Since very high data rates are transported at low signal levels, Optical Interface Area and MOST Function Area must be designed with respect to the rules of high-frequency engineering.

4.2.2 Optical Power Budget

The manufacturer of the two FOT units guarantees, that the optical power specified for the transmitter's side is emitted by the Pig Tail at the socket of the device. In addition to that he guarantees that sensitivity and dynamic of the receiver are within the specified range. The required optical power budget between sender and receiver is calculated below.

The power of the sending diode refers to the optical power after coupling in, and after 1 meter of fiber. The losses caused by coupling in need not be handled separately. The specified received power is the optical power that will actually be received by the receiving diode. Therefore no additional losses caused by decoupling need to be taken into consideration. The aging effects of the diodes and the resulting increasing attenuation are already included in the specified power values.

Worst case:

	Power (minimum)	Losses (maximum)
Minimum transmitting power in the fiber	dBm	
Tolerance of plugs for coupling in, in fiber		-----
Plug for coupler plug at device		2 dB
Fiber (10m with 0,3dB/m)		3 dB
Point of separation in door		3.5 dB
Repair		2 dB
Additional losses (By manufacturing of fiber, Assembly, Aging of fiber...)		4 dB
Plug for coupler plug at device		2 dB
Tolerance of plugs for coupling out of fiber		-----
Aging and temperature of diode		-----
Minimum received power	dBm	
<i>Difference of power/ Dynamics</i>	<i>dB</i>	
<i>Sum of losses</i>		<i>16.5 dB</i>
<i>Difference</i>	<i>dB</i>	

Table 4-1: Optical power budget in worst case, to be required between a pair of FOT units

Every pair of optical transceivers has to realize an optical power budget of at least 16.5dB, related to the power in the fiber under all conditions like variations between different examples of FOTs, aging, temperature, etc.

4.2.3 POF

Connections in the network are made by using plastic optical fiber (POF), resulting in low cost and reduced EMI problems.

4.2.4 Connection Systems (Pig Tail)

In contrast to existing systems, both transceivers are placed remotely with respect to the device's plug, e.g., on the PCB near by the MOSTTransceiver. They are connected to the device's socket with Pig Tails (Pieces of POF). This provides the following advantages:

- Possibility of protecting the end of the fiber at the device's socket.
- Easing of EMI problems.
- Flexible placement on PCB.
- Small dimensions.
- Decoupling of the plugging system of the device from the case of the FOTs.

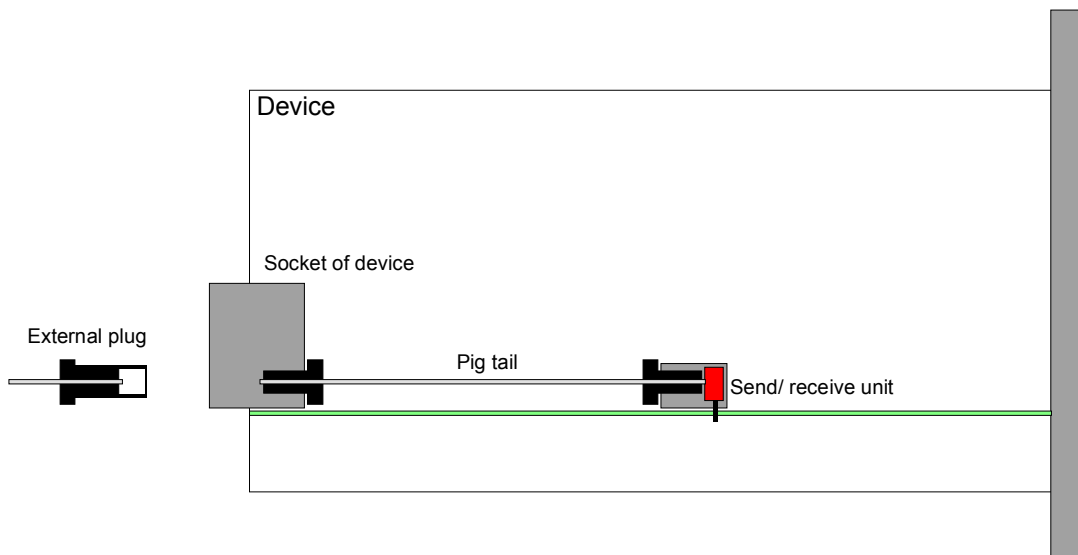


Figure 4-3: Connecting the FOTs to the plug of the device via "Pig tail"

The pig tails are connected to the FOTs and to the device's plug with a plugging system in each case.

The device's plugging system is carried out modularly and in a hybrid way. So one or more pairs of optical connectors can be combined with different electrical connectors as in a kind of model kit, for deriving plugging systems for the different devices.

4.3 MOST Function Area

The MOST function area consists of the following components (For more detailed information please refer to [7]):

- MOSTTransceiver
- Crystal
- PLL-Filter

In standalone mode, the MOSTTransceiver can communicate with I²C components (as I²C master). The transceiver itself is controlled by remote commands via the MOSTNetwork. Otherwise it communicates with a micro controller via I²C (as slave), SPI or parallel bus. Source data is exchanged with the application via the source data bus.

Reset:

On a reset, the MOSTTransceiver activates all bypass mode, switches to slave mode, and switches the interrupt pin to inactive ('1'). After reset is deactivated, the interrupt pin changes to '0' (PowerOn interrupt). The micro controller (μ C) waits for this interrupt and then initializes the transceiver in timing master or slave mode. In standalone mode, the transceiver is in all bypass mode during reset. Afterwards it auto-configures itself as slave.

For devices with μ C area it should be possible in any case, that the MOST chip can be reset by the respective μ C as well (μ C reset or Watchdog Reset), since here the MOST chip is not controlled and initialized via the network, but by the μ C.

Please note:

During Reset (not software reset), the signal RMCK is not valid. If RMCK is used as device clock, this must be taken into consideration.

4.4 μ C Area

The micro controller (μ C) area mainly consists of the μ C and some memory, and is not mandatory for a MOSTDevice (standalone mode). In the case of devices with a μ C area, there might 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 μ C must be switched off.

At the same time there are devices which must be active even if there is no network activity. Here the μ C area must be connected to a continuous power supply.

In addition to that, there are devices which are to be arranged in between. They are active without network activity, but are not connected to continuous power (for example, the power supply of a CD changer during eject of disc).

4.5 Application Area

Application area refers to the application peripherals such as receivers, amplifiers, drives, etc. The way of implementing an application area is very device-specific. In some devices, especially those with application peripherals that have high power consumption, it makes sense to supply the peripherals separately from the logic, i.e., the μC area and the MOST function area, in order to switch them on and off separately. In other applications, the application area must be connected to a continuous power supply.

If internal communication is required, the MOSTNetwork with all devices connected to it is powered. Since this may happen also in such cases where the vehicle is parked, the power consumption in this communication mode (Logic_Only_Mode) must be kept as low as possible (not only in sleep mode). This means, that it must be possible to remove the Application Area from power (if procurable).

4.6 Power Supply Area

Please note:

This section 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.

To meet these requirements, a MOSTTransceiver, 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 4-4, mainly consists of:

- Filter, unload-protection, EMI/EMC protection
- Micropower regulator (5V Cont.)
- SwitchToPower detector
- Power on logic
- Digital power supply (5V Dig)
- Application power supply (U App)
- Bad power condition comparator
- Reset generator
- Watchdog timer

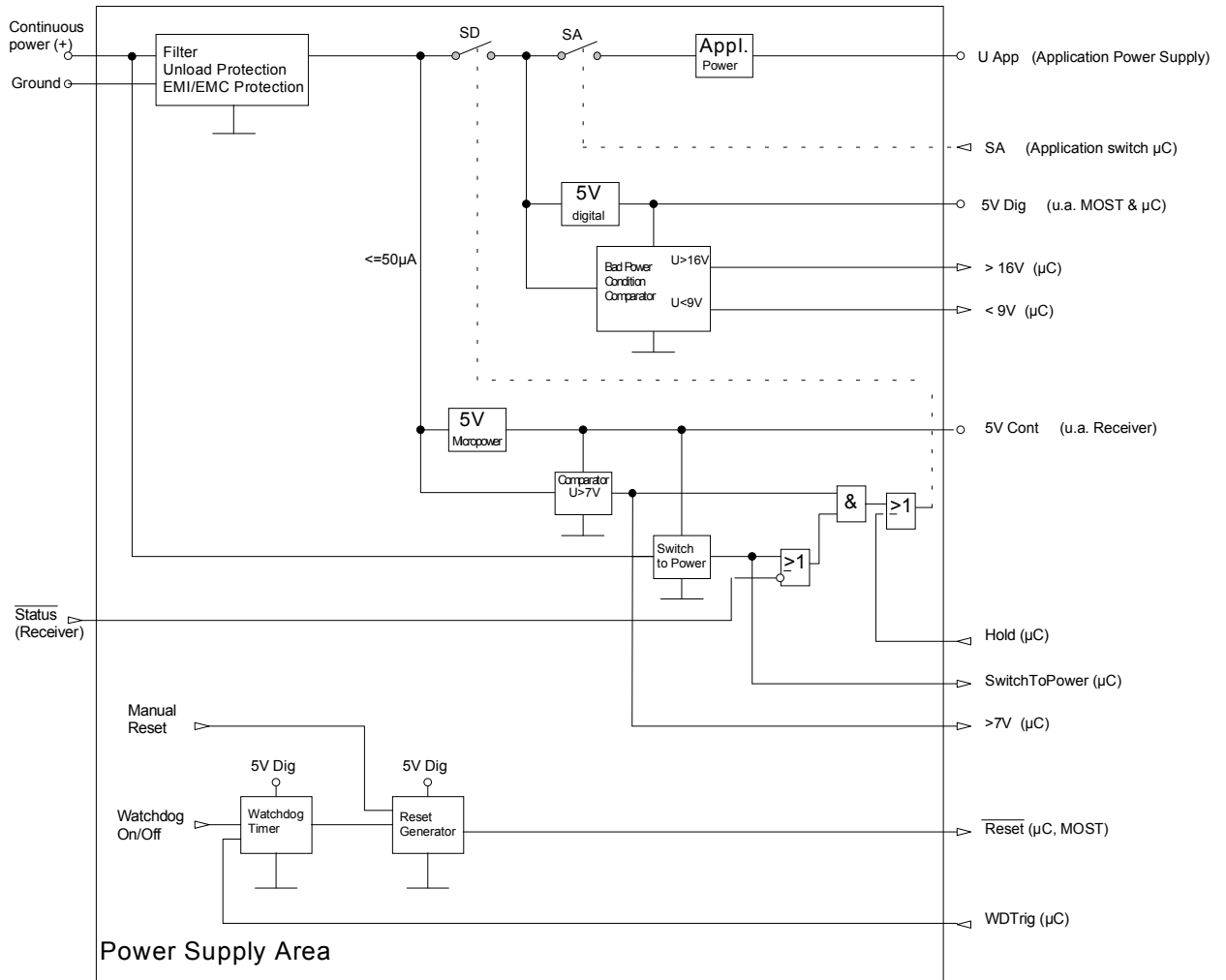


Figure 4-4: Block diagram of power supply area.

Filter and **EMI/EMC-Protection** filters the power supply and protects the device from incoming radiation, or it prevents the device from sending out radiation. The **Unload-Protection** provides the overcoming of short periods of low voltage.

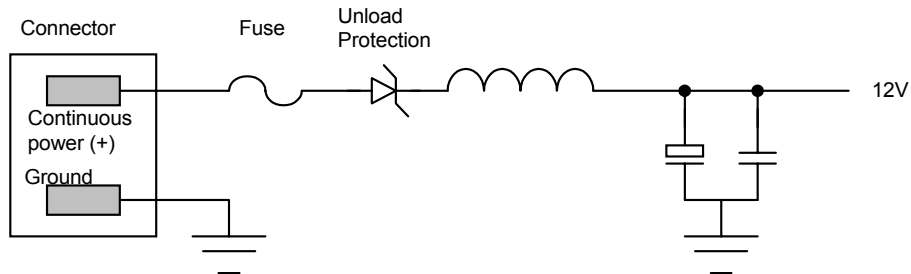


Figure 4-5: Input section of power supply area.

The **Micropower Regulator** provides power supply for the receiving FOT unit with wake-up functionality (Please refer to [6]), 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. Since the bus cannot work properly on a ring break, the devices must get a trigger in another way. Such a trigger is set by 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 ($2\text{sec} < t_1 < 4\text{sec}$). 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 t_2 , which must be long enough for the μC to safely recognize the pulse.

This is shown in the figure below for ideal signals (vertical edges). The SwitchToPower detector should be implemented at low cost, and in a way so that it works on ideal conditions as shown in the figure below. It should not be tried (at high cost) to meet the timing exactly, even on non-ideal conditions, since imprecise behavior of devices can be compensated for by the duration of the real trigger, and it is not very critical if, on an alleged trigger (e.g., caused by starting the engine), a device inadvertently switches to diagnosis mode.

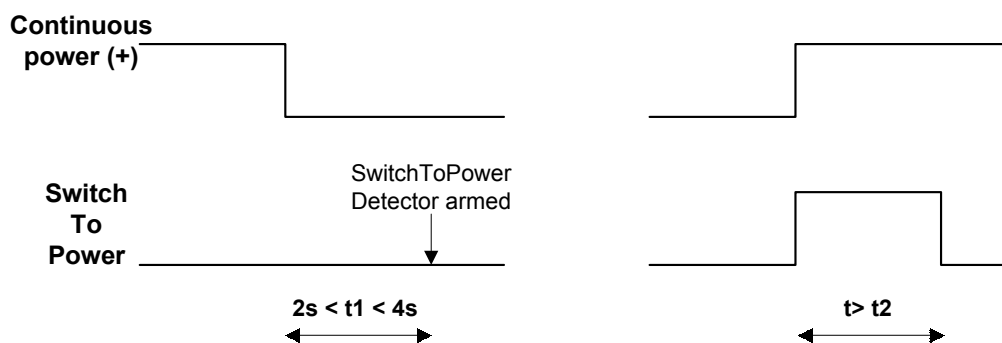


Figure 4-6: Timing of the output signal of the SwitchToPower detector depending on voltage at continuous power input.

The **Power On Logic** checks to see whether the bus is active, or if the SwitchToPower detector indicates that the device is freshly connected to power. If, in addition to that, the $U > 7\text{V}$ comparator indicates a sufficient supply voltage, switch SD is closed and **Digital Power Supply** is connected to power. **Digital Power Supply** then supplies the MOSTTransceiver 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 **7V comparator** indicates whether the input voltage is less than 7V (U_{Low}) 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 7V comparator, and of the entire digital supply voltage.

Please note:

The hysteresis must be implemented in a way, that the output signal of the 7V comparator is switched off, when the voltage drops below 7V. The output signal of the 7V comparator must then be switched on again only, if the voltage rises to $>9\text{V}$ ($+1\text{V}/-0.5\text{V}$).

The 7V 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 7V comparator must prevent the device from switching on down to a voltage level $<2V..3V$.

The **Bad Power Condition comparator** recognizes critical voltage (U_{Critical} ; $U < 9V$) and super voltage ($U > 16V$) 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 of power. The signals are evaluated only by the micro controller (μC).

The **Reset Generator** generates reset for the MOST_{Transceiver} and eventually for a μC if available. It is mandatory for all devices! Possible sources for reset are:

- Device connected to power
- Transition between low voltage to normal operation
- Low voltage on power
- Manual reset (reset button)
- Watchdog timer

The maximum length of the reset pulse is 300ms.

If a μC is available, a **Watchdog Timer** (eventually with an integrated reset generator) is mandatory. The watchdog timer initiates a reset at the reset generator, when not triggered by the μC for a certain time (WDTrig). This closes the All Bypass of the MOST_{Transceiver}. 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 Transceiver can be reset by the μC as well.

4.7 Voltage Levels

In general, a device in sleep mode must not wake the bus (light on), caused by low voltage or super voltage. Four voltage ranges are defined:

9V < U < 16V

Normal operation:

Device works normally, all functions are within the specified tolerances.

U > 16V(+/- 1V)

Super voltage:

The device is in a safe operation state, which must be defined for each device individually.

7V(+/- 0.5V) < U < 9V(+1/- 0.5V)

Critical voltage (U_{Critical}):

The device is in a safe operation state, which must be defined for each device individually. The NetInterface works normally, the device can communicate. On a recovery from this state, the network does not need to be initialized again.

U < 7 V

Low voltage (U_{Low}):

The device is in a safe operation state, which must be defined for each device individually. The voltage has dropped to a value where the device cannot communicate for long. The NetInterface does not work any longer, so a device that can not communicate safely has to switch off light in a safe way.

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 specified voltage thresholds are minimum values. This means that the application must be able to work between 9V and 16V, and that the critical voltage area reaches 7V. This does not mean that it should not been tried e.g. to enter Low Voltage at lower voltages if this is possible. Especially when using switched power supplies, it can be possible to drop the Low Voltage threshold to e.g. 3V.
2. The hysteresis ranges must be implemented for avoiding oscillating!

Low Voltage for a short period of time:

Some devices need a long time for initialization (Operating system, system communication...). If such a device would be reset even at short pulses of low voltage, it needed to be initialized after that. The interruption that would occur with respect to the entire system, would be recognizable by the customer (e.g. interruption of audio when starting the engine). Such a device should be able to survive short Low Voltage periods, without the need of being re-initialized. Especially the initialization status of the μC must be secured. This might be done, e.g., by using buffer capacitors, unload protection diode, a separate power supply for the digital section, releasing of the application peripherals, stopping the μC , etc.. Also the operation of the NetInterface can be reduced, e.g. by resetting the MOST Transceiver, which will then close its All Bypass (Except a device containing the timing master). The light should be kept switched on as long as possible, since then the rest of the system would not be disturbed. After the Low Voltage period the MOST Transceiver 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 124.

5 Tools

5.1 OptoLyzer4MOST[®]

OptoLyzer4MOST[®] of Oasis SiliconSystems is an optical analysis tool for the MOSTNetwork. It provides the following features:

1. **Viewer:**
It is possible to open several viewer windows, to conveniently analyze the data traffic on the control channel of the bus, either online or offline. Messages are automatically disassembled based on the syntax tree.
2. **Syntax Tree:**
Telegrams for controlling devices can be assembled (sending) or disassembled (receiving) by a syntax tree. The syntax tree is user definable (text based) and is compiled via the syntax compiler.
3. **Filter:**
Relevant information can be extracted based on a variety of possible adjustments.
4. **Recorder:**
Traffic on the control channel can be recorded and saved.
5. **Sender:**
Control messages can be sent via the bus. They can be assembled based on the syntax tree.
6. **Remote Control:**
Panels for remote controlling of devices can be generated, which can simplify repetitive operations.
7. **Timing Analysis:**
Analysis of traffic and response times of devices on the bus.
8. **Source Data Control:**
Various possibilities for influencing channels in the synchronous area.
9. **Synchronous Interface:**
Several analog and digital I/Os for synchronous signals at the OptoLyzer4MOST[®] Interface Box.
10. **Stress Functions:**
Bus load generation and error simulation on the MOSTNetwork.
11. **Spy Mode:**
Viewing the entire traffic on the control channel of the MOST bus by an OptoLyzer4MOST[®] Interface Box that is "invisible" for devices at the bus.

The OptoLyzer4MOST[®] system consists of hardware (OptoLyzer4MOST[®] Interface Box), which is connected via RS232 to a PC, and PC software, which provides an interface for other applications.

5.2 MOSTRapidControl

The software tool MOSTRapidControl is based on the OptoLyzer4MOST[®] PC software. It provides fast and effective generation of operating and display panels, which can be individually designed to meet a wide range of needs. It is possible to assign commands to operation elements to be activated on certain operations. Display elements can filter the relevant telegrams out of the data stream. In addition to that, functions can be defined to translate the sequence of bits in the telegrams to a selected display format. The syntax tree of the OptoLyzer4MOST[®] PC software can also be used.

5.3 OptoLyzer4MOST[®] Professional

5.3.1 Introduction

5.3.2 Architecture

The basis of OptoLyzer4MOST[®] Professional is the OptoLyzer[®] Socket Library. It consists of a collection of library functions which cover all aspects of analysis and simulation of a MOSTNetwork. OptoLyzer[®] Socket Library has a generic interface for the hardware. Adapting to different OptoLyzer hardware components is done by device-specific libraries (DLLs), which can be integrated into the system as needed, e.g., together with new hardware.

Device-specific DLLs are provided for connecting the OptoLyzer4MOST[®] Interface Box via RS232 and for PC-Boards via ISA/ PCI bus. The structure of the OptoLyzer[®] Socket Library is based on three main functional blocks:

- OptoLyzer control block
- Bus analyzer
- Simulation interface

5.3.3 OptoLyzer Control

The OptoLyzer control block connects the OptoLyzer4MOST[®] hardware to the OptoLyzer control software. This block administers global settings such as the usage of PC interfaces. In addition to that, the specific properties of the respective OptoLyzer4MOST[®] Interface Boxes are mapped, e.g., codec, signal processing, mapping of PC sound channels, Wake-Up, etc.

These properties can be visualized with control windows, which are partly integrated in the basic software. Other control windows are based on independent executables (control modules), which can be added to the system later.

Control modules available to the system are automatically found and integrated in the task bar, so that the user sees one integrated system.

Programs listed below are examples of plug-ins which can be integrated into the system:

- MOSTEDIT, an easy-to-use register editor for the MOSTTransceiver OS8104.
- MOSTRadar, an analysis tool for exploring every MOSTNetwork in detail.
- Oasis Flashwriter, a tool for automatic firmware updates for OptoLyzer4MOST[®] Interface Boxes and all connected MOSTDevices.

New hardware developed for MOST can be integrated with the system by providing a system-specific DLL and a new control window. Since the system is very flexible, it can accommodate new abilities of the hardware as well.

5.3.4 Bus Analysis

This function block collects bus events, such as control messages, lock, unlock, light, or error conditions. All events are provided with a time stamp and collected in a central message buffer. Bus analysis provides several functions with a uniform interface to the visualization software, including filtering functions, functions for recording events, and functions for generating trigger conditions.

The analysis windows for viewing data are also based on the OptoLyzer[®] Socket Library. By default, the “classic” analysis windows like Viewer, Timing Analyzer and Recorder are built in. In addition to that, modules can be added later for more detailed analysis.

This makes it possible to generate complex recording modules with pre- and post-triggering, or specific viewers including special disassembling functions. A very interesting application would be to add recording modules that evaluate and record events on the MOSTNetwork and special data at the same time. This way it would be possible, e.g., to record the audio output of a tuner together with the respective messages and events on the MOSTNetwork. Of course, the hardware environment must meet the requirements of this application.

Since the bus analysis function block is closely associated with the simulation interface, messages from the virtual MOSTNetwork can also be recorded.

5.3.5 MOST Simulation Interface

When developing applications for MOSTDevices, simulating the application in a virtual MOST environment helps to optimize the development process. A simulation can be used for testing before the actual hardware is available. The optimal approach for simulation is to combine simulation of virtual MOSTDevices with real-world devices. It is implemented in this way in OptoLyzer4MOST[®] Professional.

OptoLyzer4MOST[®] Professional consists of a collection of library functions covering all aspects of analysis and simulation of a MOSTNetwork. OptoLyzer[®] Socket Library has a generic interface for the hardware. Adapting to different OptoLyzer hardware components is done by device-specific libraries (DLLs), which can be integrated into the system subsequently (e.g., together with new hardware).

An application written in ANSI-C can be simulated just by adding a few functions that handle connection to the OptoLyzer[®] Socket Library. When implementing the simulated application into hardware, the simulation-relevant functions can be disabled by removing a single switch.

The simulation function of OptoLyzer4MOST[®] Professional covers all steps of development of a MOST system. It is possible to simulate virtual MOSTDevices communicating with other virtual MOSTDevices in a virtual MOSTNetwork. In addition to that, it is possible to mix real and virtual MOSTDevices, so that virtual MOSTDevices can communicate with real devices.

It is the simulation interface that links the virtual and the real MOSTNetworks. It provides a message interface, which can support almost any number of virtual MOSTDevices. The interface itself consists of functions that provide sending and receiving of MOST messages and the handling of group address and logical address. For a simulated MOSTDevice, the OptoLyzer[®] Socket Library behaves like a real MOST node. In addition to that, a real MOSTTransceiver can be assigned to a virtual MOSTDevice in an OptoLyzer4MOST[®] Interface Box or a PC board. The simulated MOSTDevices become “visible” and addressable for real MOSTDevices.

The simulation interface functions are equivalent to the functions of the basic layer (Layer I) of the MOSTNetServices API, which are a library implemented in ANSI C providing software access to the MOSTNetwork.

This ability to simulate virtual devices in a virtual network based on the OptoLyzer[®] Socket Library allows the interaction between MOSTDevices to be tested before they have been developed. This is a very useful feature when defining new systems.

With the OptoLyzer[®] Socket Library, several PC boards or OptoLyzer4MOST[®] Interface Boxes can be connected to cover MOST systems of any complexity. Since the OptoLyzer[®] Socket Library offers a variety of interfaces, it is no problem to find an appropriate development platform.

6 Appendix A: Index of Figures

Figure 2-1: Model of a MOSTDevice	14
Figure 2-2: Communication with a function via its function interface (FI).....	14
Figure 2-3: Structure of a function block consisting of functions classifiable as methods, properties and events.....	16
Figure 2-4: Setting a property (Temperature setting of a heating)	17
Figure 2-5: Reading a property (Temperature setting of a heating).....	18
Figure 2-6: Status report of property temperature setting	18
Figure 2-7: Example for a function interface (FI).....	19
Figure 2-8: Ideal audio system	23
Figure 2-9: Real audio system.....	24
Figure 2-10: Delegation of functions of all audio components to one audio controller.....	24
Figure 2-11: Highest layer of the device model.....	26
Figure 2-12: Device model for audio sources with player function.....	27
Figure 2-13: Device model for audio sources without player function.....	28
Figure 2-14: Processing of messages including error check on different layers.....	36
Figure 2-15: Sequences when using Start with and without error.....	37
Figure 2-16: Flow for handling communication of methods (slave's side).....	38
Figure 2-17: Flow for handling communication of methods (controller's side).....	38
Figure 2-18: Virtual communication between two devices on application layer and real comm. via network.....	44
Figure 2-19: Device with MOST address CDC, a function block CD Player with FBlockID CD, and its functions.....	45
Figure 2-20: Communication between two devices via the different layers	46
Figure 2-21: Example for a slave device	47
Figure 2-22: Virtual illustration of the controlled properties in the control device.....	48
Figure 2-23: Unambiguous assignment between protocol and variable.....	49
Figure 2-24: Controlling multiple devices.....	50
Figure 2-25: Controlling two identical devices.....	51
Figure 2-26: Hierarchical structure of the protocol filter (command interpreter).....	52
Figure 2-27: Routing answers in case of multiple tasks (in one controller) using one function.....	53
Figure 2-28: reading the function blocks of a device from NetBlock	55
Figure 2-29: Requesting the functions contained in an application block.....	56
Figure 2-30: Requesting the function interface of a function	57
Figure 2-31: Meaning of position x in record (above) and of position y in a record with array (below).....	67
Figure 2-32: Position x in case of an array of basic type (left), and y in case of an array of record (right).....	68
Figure 3-1: Structure of blocks and frames on the MOST bus.....	86
Figure 3-2: Layer model of a device	96
Figure 3-3: Flow chart "Overview of the states in NetInterface"	98
Figure 3-4: Behavior of a master device in state NetInterfacelnit.....	100
Figure 3-5: Behavior of a waked slave device in state NetInterfacelnit.....	101
Figure 3-6: Behavior of a waking slave device in state NetInterfacelnit.....	102
Figure 3-7: Behavior in state NetInterfaceNormalOperation.....	104
Figure 3-8: Localizing a fatal error with the help of ring break diagnosis.....	105
Figure 3-9: Behavior during ring break diagnosis in a timing master (part 1).....	107
Figure 3-10: Behavior during ring break diagnosis in a slave (part 1).....	108
Figure 3-11: Behavior during ring break diagnosis in a timing master and slave (part 2).....	109
Figure 3-12: Behavior during ring break diagnosis in a timing master and slave (part 3).....	110
Figure 3-13: Flow of initialization on application level in a network slave.....	112
Figure 3-14: Flow of initialization on application level in a network master.....	115
Figure 3-15: Flow in network master during requesting system configuration.....	116
Figure 3-16: Example (2 devices) for waking of the MOST network via light on the network.....	117

Figure 3-17: Switching off MOST network via starting method ShutDown in every NetBlock, and signaling to every application, and switching off light..... 118

Figure 3-18: Prevention of switching off MOST network via ShutDown.Result (Suspend)..... 119

Figure 3-19: Behavior of a device depending on supply voltage..... 125

Figure 3-20: Seeking the logical address of a communication partner 133

Figure 3-21: Possible mechanism to adapt transfer rates to the speed of a data sink during segmented transfer..... 134

Figure 3-22: NetServices: Services for control channel 135

Figure 3-23: Sending a message on the control channel (Low level). 137

Figure 3-24: Remote read..... 139

Figure 3-25: Remote write. 140

Figure 3-26: NetServices for the synchronous channel..... 141

Figure 3-27: NetServices: Services for the asynchronous channel..... 152

Figure 3-28: Flow of building a connection in synchronous area. 156

Figure 4-1: Structure of a MOSTDevice; The different functional areas and their interfaces..... 161

Figure 4-2: Optical interface area. 162

Figure 4-3: Connecting the FOTs to the plug of the device via "Pig tail" 165

Figure 4-4: Block diagram of power supply area. 168

Figure 4-5: Input section of power supply area. 169

Figure 4-6: Timing of the output signal of the SwitchToPower detector depending on voltage at continuous power input..... 170

7 Appendix B: Index of Tables

Table 2-1: Ranking of device classes	25
Table 2-2: Application example for the principle of derived devices	27
Table 2-3: FBlockIDs.	30
Table 2-4: OPTypes for properties and methods	33
Table 2-5: Error codes and additional information.....	34
Table 2-6: Classes of functions with a single parameter.	58
Table 2-7: Available units.....	62
Table 2-8: Notification matrix (x=notification activated).....	81
Table 2-9: Parameter Control	82
Table 2-10: Protocols with different controls for making entries in the notification matrix, and the resulting entries.	82
Table 3-1: Structure of the MOST frame	86
Table 3-2: Structure of a frame in the asynchronous area (48 bytes data link layer).....	89
Table 3-3: Structure of a frame in the asynchronous area (alternative data link layer).....	90
Table 3-4: Structure of a control data frame.....	91
Table 3-5: Addressing modes vs. address range	93
Table 3-6: Events in state NetInterfacePowerOff	99
Table 3-7: Events in state NetInterfaceInit.....	99
Table 3-8: Events in state NetInterfaceNormalOperation.....	103
Table 3-9: Events in state NetInterfaceRingBreakDiagnosis.....	105
Table 3-10: Functions in conjunction with power management.	119
Table 3-11: KWP2000 on MOST. Overview of identifiers.	126
Table 3-12: Functions in NetBlock that handle addresses	128
Table 3-13: Priorities on the control channel	129
Table 3-14: Example for a De-central Registry.	130
Table 3-15: Central Registry.....	131
Table 3-16: Commands in conjunction with the Central Registry.....	132
Table 3-17: Functions for building a De-central Registry if a Central Registry is available.....	132
Table 3-18: Functions in NetBlock in conjunction with the administration of synchronous resources.	142
Table 3-19: Common functions in a function block for administering synchronous resources	143
Table 3-20: Functions in a function block with a synchronous source, in conjunction with administering synchronous resources.	146
Table 3-21: Functions in a function block with a synchronous sink in conjunction with the administration of synchronous resources.....	147
Table 3-22: Functions for administration of boundary between synchronous and asynchronous area in network master.	153
Table 3-23: Functions in connection master in conjunction with the administration of synchronous connections.....	155
Table 3-24: Timeouts	160
Table 4-1: Optical power budget in worst case, to be required between a pair of FOT units	164

8 Appendix C: INDEX

μ	
μC	166
μC Area.....	161, 166
7	
7V Comparator	170
A	
AbilityToWake	117, 119, 120
Abort	40
Absolute	78
ABY	84
ACK	12
Activity Detection	162
Addressing	
Modes (OS8104).....	93
Network.....	127
OS8104.....	93
Physical Address	130
ADS	151
All Bypass	84, 171
Allocating Synchronous Channels	94
Allocation	94
AMS	135
Application	
Hanging	126
Application (init)	111
Application Area.....	161, 167
Application Error	35
Application Message Service	135
Area	
Application Area.....	161, 167
Micro Controller Area	161, 166
MOST Function Area	161, 166
Optical Interface Area	161, 162
Power Supply Area	161, 167
Array	65, 68
Array Window.....	74
Mother Array	73
Selecting In	70
Array Window.....	74
Asynchronous Channel.....	13
Asynchronous Data.....	88, 89
Asynchronous Data Transmission Service	151
B	
Bad Power Condition Comparator	171
Bandwidth	153
Bandwidth (Synch. / Async.).....	13
Block	85
bMPR.....	103
Bool-field.....	41
Bottom	76
Boundary	87, 153
Boundary Descriptor	13, 87, 153

Broadcast.....	93
Broadcast Address.....	127, 128
Broadcast.Configuration.Status	111
bSBC	87
BuildSyncConnection.....	154
bXRTY	92
bXTIM	91
Bypass	
All Bypass	84, 171
Source Data Bypass	84

C

Catalog	43
Central Registry	55, 113, 114, 131
Channel	95
Unused	95
Channel Allocation	94
Class.....	59
List	59
CMS.....	135
Communication	44
Comparator	
7V	170
Configuration.Status	111, 113
Connection Master.....	154
Control Channel.....	12
Control Data Interface.....	91
Description	91
Frame	91
Control Message	
Addressing (OS8104)	93
Receiving (OS8104).....	138
Sending (OS8104)	137
Control Message Service.....	135
Control Messaging Interface	91
Description	91
Frame	91
Controller	15
CRA	94
CRC	12, 89, 138
CreateArrayWindow.....	74
Critical Voltage.....	124, 171
Crystal.....	166

D

Data	41
Data Link Layer.....	151
Data Transport In A MOST System	85
Data Types	
Bool-field.....	41
Enum	41
Signed Byte.....	41
Signed Long.....	41
Signed Word	41
Unsigned Byte.....	41
Unsigned Long.....	41
Unsigned Word	41
De-Central Registry	130
Decrement	40
Delay.....	94
Delay Compensation.....	149
Delegation.....	23
Deriving Devices	26
DestroyArrayWindow	74

Device.....	14
Device Hierarchy.....	26
DeviceID	29
DeviceNormalOperation.....	97
DevicePowerOff.....	97
DeviceStandBY.....	97
Diagnosis	105, 126
Diagnosis Error Shut Down.....	105
Diagnosis Ready.....	105
DiagnosisStart	99
Digital Power Supply.....	170
Down.....	77
Dynamic Array	71
Dynamic Behavior.....	96

E

Ethernet	151
Electrical Bypass.....	84
EMI/EMC-Protection	169
Enum	41
Enumeration	58, 64
Error.....	34, 37, 39, 40
Application Error	
Error Secondary Node	35
General Execution Error	35
Parameter Error	35
Specific Execution Error.....	35
Temporarily Not Available Error.....	35
Fatal Error (Network)	121, 122
Handling In Connection Master.....	154
Managing Errors (Network).....	121
Network Change Event (Network).....	121, 124
Syntax Error	34
Unlock (Network)	121, 123
Voltage Low (Network).....	121, 124
Error Checking (Flow Chart)	36
Error Secondary Node	35
Error Shut Down	103
ErrorAck.....	39
Event.....	16, 18
Diagnosis Error Shut Down.....	105
Diagnosis Ready.....	105
DiagnosisStart.....	99
Error Shut Down	103
Init Error Shut Down.....	99
Init Ready.....	99
Net On	103, 111
NetworkChange	111, 113
Normal Shut Down	103
StartUp.....	99
Exponent.....	42

F

Fatal Error (Network)	121, 122
FBlock.....	15, 29
FBlockID	15, 29, 30
List	30
FBlockIDs	55
FI 19	
Filter.....	169
FkID	15, 29, 32
Ranges	32
FkIDs	56
Flags.....	58

FOT.....	162
Frame	85, 91
Function	16
In Documentation.....	43
Function (Fkt).....	15, 29
Function Block.....	14
Function Catalog.....	43
Function Class	
Array	68
Dynamic Array	71
Enumeration.....	58, 64
For Methods.....	80
Long Array	73
Number	58, 61
Record	66
Switch	58, 60
Text.....	58, 63
Function Interface	14, 19, 57

G

General Execution Error	35
Get	39
GetInterface	40
Group Address.....	127, 128
Groupcast	93

H

Hardware Design Rules (Optical Interface).....	163
Heredity	25
High Level Retries.....	129
Hold Mechanism	171
Hysteresis	170

I

I ² C.....	166
Increment.....	40
Init Error Shut Down.....	99
Init Ready.....	99
InstID	31
Interface.....	40
Interface For Synchronous Source Data.....	88

K

KWP2000.....	126
--------------	-----

L

Layer Model	96
Length.....	41
Light Off	121
Lock	
Stable.....	99, 101, 106
Logic_Only_Mode	167
Logical Address	127, 133
Long Array	73
Low Level Retries	92, 129
Low Power (OS8104).....	95
Low Voltage	124, 170

M

Master	84
Network Master	113
mCRA	94
Message Notification	81
Message Sink	
Overload	134
Method	16
Micro Controller Area	161, 166
Micropower Regulator	169
MMI	15
More information	2
MOST	11
MOST Device	14, 96
MOST Frame	85
Boundary Descriptor	87
Preamble	87
Structure Of	86
MOST Function Area	161, 166
MOST High Protocol	136, 152
MOST System Services	23
MOST Transceiver	84
MOSTRapidControl	175
MostSetBoundary(Timing Master)	142
Mother Array	73
MPR	103

N

NAK	12, 134
Name	59
NetInterface	96, 98
NetInterfaceInit	99
NetInterfacePowerOff	99
NetOnEvent	103, 111
NetServices	96
Network	
Switching Off	117
Waking	117
Network Change Event (Network)	121, 124
Network Master	113
Network Slave	111
NetworkChange	111, 113
Node Address	127
Node Position	93
Node Position Address	94, 127
Normal Shut Down	103
NormalOperation	103
Notification	16, 81
Notification Matrix	81
Null Termination	42
Number	58, 61

O

Object	32
Operation	122
Operation Type	15, 29
Optical Interface Area	161, 162
Optical Power Budget	163, 164
OptoLyzer4MOST	174
OptoLyzer4MOST Professional	176
OptPwrSwitch	163
OPType	15, 29, 33
List	33

OPTypes	59
OS8104	84
Overload In A Message Sink	134

P

Packet Data	88, 89
Accessing (OS8104)	150
Parameter Error	35
Phase Jitter	163
Physical Position	127
Physical, Link and Transaction layers	22
Pig Tail	165
PLL	166
POF	164
Position	66
Power Management	
Administration	117
OS8104	95
Power Master	117, 118
Power On Logic	170
Power Save Mode	124
Power Supply Area	161, 167
Preamble	87
Priority Levels	
Control Channel On Network Level	129
Packet Data Transfer	150
Processing	37
ProcessingAck	39
Properties With Multiple Variables	65
Property	16
Reading	18
Setting	17

Q

Query	118
-------------	-----

R

Ranking	25
RE	89
Receiving Control Messages (OS8104)	138
Record	66
Registry	
Central Registry	113, 114, 131
De-Central Registry	130
Remote Access	94, 139
Remote Read Message	139
Remote Write Message	140
Reset Generator	171
Result	37
ResultAck	39
Retries	
High Level Retries	129
Low Level Retries	92, 129
Retry Time	91
Ring Break Diagnosis	105
RLE	56
Routing Engine	89
Run Length Encoding	56

S

SAI	93
SBC	87

SBY.....	84
SearchAW.....	79
Secondary Node.....	35
Securing Data.....	151
Seeking Logical Address.....	133
Segmented Transfer.....	135
Selecting In Array.....	70
Sending Control Messages (OS8104).....	137
Set.....	39
SetGet.....	39
Shadow.....	47
ShutDown.....	118, 119, 120
Sign.....	41
Signed Byte.....	41
Signed Long.....	41
Signed Word.....	41
Simulation.....	177
Single Transfer.....	135
Slave.....	15, 84
Network Slave.....	111
Non Waking.....	100
Sleep Mode.....	161
Source Data.....	88
Bypass.....	84
Handling In Function Block.....	143
Handling In Function Block (Synchronous Sink).....	146
Handling In Function Block (Synchronous Source).....	143
Handling In NetBlock.....	142
Handling In NetServices.....	141
Interface.....	88
Parallel Interface (OS8104).....	149
Serial Interface (OS8104).....	149
SourceHandles.....	142
SourceInfo.....	143
Specific Execution Error.....	35
SPI.....	166
SR1.....	88
Stable Lock.....	99, 101, 106
Start.....	37
Start Address Initialization.....	93
StartAck.....	39
StartResult.....	37
StartUp.....	99
State.....	98
NetInterfacelnit.....	99
NetInterfacePowerOff.....	99
NormalOperation.....	103
Status.....	39, 162
Step.....	42
String.....	42
Structure Of Data Packet.....	
48 Bytes Data Link Layer.....	89
Alternative Data Link Layer.....	90
Structure Of MOST Frame.....	86
Super Voltage.....	171
Switch.....	58, 60
Switching Off Network.....	117
SwitchToPower.....	105
SwitchToPower Detector.....	169
SX1.....	88
SyncAlloc(Source).....	141
SyncConnectionTable.....	155
SyncDeAlloc(Source).....	141
SyncFindChannels.....	142
Synchronous Channel.....	12, 89
Synchronous Connection.....	

Establishing.....	156
Removing.....	158
Supervising.....	158
Synchronous Source Data.....	88
SyncOutConnect(Sink).....	141
SyncOutDisconnect(Sink).....	141
Syntax Error.....	34
SystemCommunicationInit.....	111
SystemCommunicationInit.....	112

T

tAnswer.....	159
tCfgStatus.....	159
tConfig.....	159
TCP.....	136
tDiag_Lock.....	160
tDiag_Master.....	106, 160
tDiag_Slave.....	106, 160
TDM.....	89
Temporarily Not Available Error.....	35
Text.....	58, 63
Time Division Multiplexing.....	89
Timeouts.....	159
Timing Master.....	84
MostSetBoundary.....	142
Timing Slave.....	84
tLock.....	99, 100, 106, 159
tMaster.....	99, 122, 159
tMsgResponse.....	160
Tools	
MOSTRapidControl.....	175
OptoLyzer4MOST.....	174
OptoLyzer4MOST Professional.....	176
Top.....	76
tOptPwrLow.....	160, 163
tPowerOn.....	159
tPwrSwitchOffDelay.....	118, 121, 160
Transparent Channels.....	88
tRestart.....	117, 118, 121, 122, 160
tRetryShutDown.....	118, 159
tShutDown.....	159
tShutDownWait.....	118, 159
tSlave.....	99, 122, 159
tSuspend.....	118, 159
tUnlock.....	103, 160
tUnlockRecovery.....	103, 160
tWaitAfterNCE.....	159
tWaitNodes.....	159
tWakeUp.....	159
TXR.....	93

U

Unclassified Method.....	59
Unclassified Property.....	59
Unicode.....	42
Unit.....	42, 61
List.....	62
Unload-Protection.....	169
Unlock (Network).....	121, 123
Unsigned Byte.....	41
Unsigned Long.....	41
Unsigned Word.....	41
Up.....	77

V

VehicleManufacturerECUHardwareNumber	126
Virtual Communication	44
Voltage	
Critical Voltage	124, 171
Critical Voltage Range	172
Handling Low Voltage	173
Low Voltage	124
Low Voltage Range	172
Normal Operation Voltage Range	172
Super Voltage	171
Super Voltage Range	172
Voltage Low (Network)	121, 124

W

Waking	122
Waking Of The Network	117
Watchdog	126
Watchdog Timer	171
WDTrig	171

X

Xmit Retry Time	91
XRTY	92

Z

Zero Power (OS8104)	95
---------------------------	----

Notes:

