MOST

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

Multimedia and Control Networking Technology

Recommended Communication Profile Application Note

Rev. 1.0

07/2012

MOSTCO CONFIDENTIAL

See page 3 for the terms of disclosure

MOST® COOPERATION

Legal Notice

COPYRIGHT

LICENSE DISCLAIMER

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

CONTENT AND LIABILITY DISCLAIMER

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

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

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

FEEDBACK INFORMATION

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

TRADEMARKS

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

SUPPORT AND FURTHER INFORMATION

For more information on the MOST technology, please contact:

MOST Cooperation
Administration
Bannwaldallee 48
D-76185 Karlsruhe
Germany

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

E-mail: contact@mostcooperation.com
Web: www.mostcooperation.com

Contents

Document History

Rev. 1.0

Change Ref.	Section	Changes
1V0_001	General	Initial Version.

Bibliography

All documents, which are referenced by this MOST document, are listed here along with their versions. For the current release status please refer to the MOST Cooperation Document List.

	Document	Revision
[1]	**MOST Specification**	3.0

Table Bibliography-1: Document references

1 General

1.1 Storage

1.2 Abbreviations

Abbreviation	Description
RPC	Remote Procedure Call
UTF	UCS (Universal Character Set) Transformation Format
GB	Guojia Biaozhun
FBlock	Function Block or Server in Client-server model terminology
OPType	Operation Type
FktIDs	Function IDs

Table 1-1: Abbreviations

1.3 Glossary

Term	Description
GB 18030	Chinese Encoding Standard from 2000 for the support of its national characters.
Function Block	Represents Server in Client-server model.
Shadow	Represents Client in Client-server model.
Operation Type	Gives message type more context.
Function IDs	Wellknown function implemented by each Function Block. Delivers all functions implemented by Function Block.

Table 1-2: Glossary

1.4 Purpose

The Recommended Communication Profile specifies a limited set of MOST protocol mechanisms. The reasons for doing this are the following:

a) Remove mechanisms which are not needed

b) Remove very complicated protocol parts

c) Try to close functional gaps

d) Remove ambiguous mechanisms

In other words, the Recommended Communication Profile tries to reduce itself to a subset of the currently existing MOST mechanisms, which provide all necessary mechanisms for building distributed infotainment systems.

This document serves as guideline that describes the proposed subset and also describes how removed mechanisms can be compensated. The document is focused on following major MOST mechanisms:

a) Addressing

b) MOST wire format (serialized format): Data Types, Function Classes, Operation Types

c) RPC mechanisms

The intention is to define a set of rules and mechanisms but at the same time keeping MOST Specification 3.0 E2 [1] compatibility. Any device which uses the Recommended Communication Profile must pass MOST Compliance testing.

2 MOST Data Types

This chapter goes through all MOST Data Types which will be used inside the Recommended Communication Profile.

The following data types will be explicitly removed and will not be discussed in the following sections:

a) BitField –This functionality can be built on top of the existing number types.

b) Enum – Instead, one of the six number data types can be used. This gives also additional benefit in case enumeration uses more than 256 elements.

c) Classified Stream – This construct can be built by using other mechanisms.

This Recommended Communication Profile document distinguishes between scalar (base) data types and extended data types. Scalar data types can be used inside other containers to build MOST messages. Extended data types use scalar (basic) types for constructing payload.

The Recommended Communication Profile supports only one extended data type called MOST Stream.

1.1 Scalar MOST Types

2.1.1 MOST Boolean

The Recommended Communication Profile specifies MOST Booleans as follows:

a) FALSE – Will be represented by 0.

b) TRUE – Represented by any other value from 0x01 to 0xFF.

2.1.2 MOST Integer Data Types (Byte, Word and Long)

All existing integer data types are kept. The following six data number types will be supported:

Data Type	Size in bytes	Value Range
Unsigned Byte	1	0 to 255
Signed Byte	1	-128 to 127
Unsigned Word	2	0 to 65535
Signed Word	2	-32768 to 32767
Unsigned Long	4	0 to 4294967295
Signed Long	4	−2147483648 to 2147483647

Table 2-1: MOST Integer Data Types

All signed integer data types shall be interpreted as two's complement. The following is an example for 8 bit two's complement notation:

Binary value	Two's complement interpretation
00000000	0
00000001	1
…	…

Binary value	Two's complement interpretation
01111110	126
01111111	127
10000000	-128
10000001	-127
10000010	-126
…	…
11111110	-2
11111111	-1

Table 2-2: Two's complement interpretation

2.1.3 MOST Strings

The MOST String data type will be limited to only UTF-8 encoding. Please note that UTF-8 is also compatible with ASCII representation.

See the following example with UTF8 encoding used inside MOST String:

Identifier	Content
UTF-8	“Hello, world!\0”
0x02	48 65 6C 6C 6F 2C 20 77 6F 72 6C 64 21 24 00

Table 2-3: UTF8 String Encoding Example

This makes string parsing easier because even without knowing MOST String representation, one could just read up to the first zero terminator which terminates the string. Afterwards, the next MOST parameter follows.

GB 18030-2005

GB 18030 is a new Chinese code page standard that extends GB 2312-1980 and GBK. GB 18030 is a multi-byte encoding using 1, 2, and 4-byte codes (similar to UTF-8). It is mandatory to support GB 18030 for all software sold in China. It is still possible to use Unicode encoding internally (communication, internal inter process communication, etc.) and to support GB 18030 at text input and output points (text fields, text forms, etc.). That means to support GB 18030, only a conversion between GB 18030 and ISO 10646/Unicode is needed. This is possible because of the definition of GB 18030 with a mapping table to ISO 10646/Unicode.

2.2 Extended MOST Types

2.2.1 MOST Stream

The Recommended Communication Profile only uses the following MOST stream types:

a) Unstructured Stream

b) Simple Stream

3 MOST Communication

3.1 Session Handling

Session handling is added so that connection-oriented communication is fully supported.

The following MOST functions will be provided by each FBlock to support sessions:

Function	Function Name	Operation Type	Paramaters
0xC00	CreateSession	StartResultAck (0x06)	ClientID
		ErrorAck (0x09)	ClientID, ErrorCode, ErrorInfo
		ResultAck (0x0D)	ClientID, SessionID
0xC01	CloseSession	StartResultAck (0x06)	SessionID
		ErrorAck (0x09)	SessionID, ErrorCode, ErrorInfo
		ResultAck (0x0D)	SessionID

Table 3-1: Session handling functions

Parameter description:

Parameter Name	Data Type	Parameter Description
ClientID	Unsigned Word	Used only for creating a session. Needed to be able to distinguish between two shadows with same ID (FBlockID.InstanceID) in one device. For example, it could be that two shadows with the same ID in one device send CreateSession Method to an FBlock. The FBlock cannot distinguish between these two Methods without ClientID. The Shadow is responsible for ClientID uniqueness. The following mechanisms are proposed to make ClientID unique: a) Shadow resides in OSEK like system: OSEK Task ID could be used. b) Shadow resides in dynamic OS system which supports Processes and Threads: ProcessID or ThreadID could be used (or a combination of the two). c) Shadows use dedicated Middleware: Client handle from existing Middleware infrastructure could be used.
SessionID	Unsigned Word	SessionID will be generated by the FBlock in the case that the CreateSession function succeeds. SessionID is incremented with each new session (pseudo randomized SessionIDs could be used for secure applications).

Table 3-2: Session handling parameters

A session does not expire automatically. A session is invalidated, for example, in these situations:

a) The Shadow detects that the FBlock is not visible – the Shadow terminates the session.

b) The Shadow terminates the session automatically.

c) The FBlock receives CreateSession from a Shadow which already opened a session – the old session is terminated on the FBlock side and a new session is generated.

SessionID will be used in all subsequent Methods as Shadow indicator.

See the following sequence diagram with two Shadows in one device and FBlock in separate one.

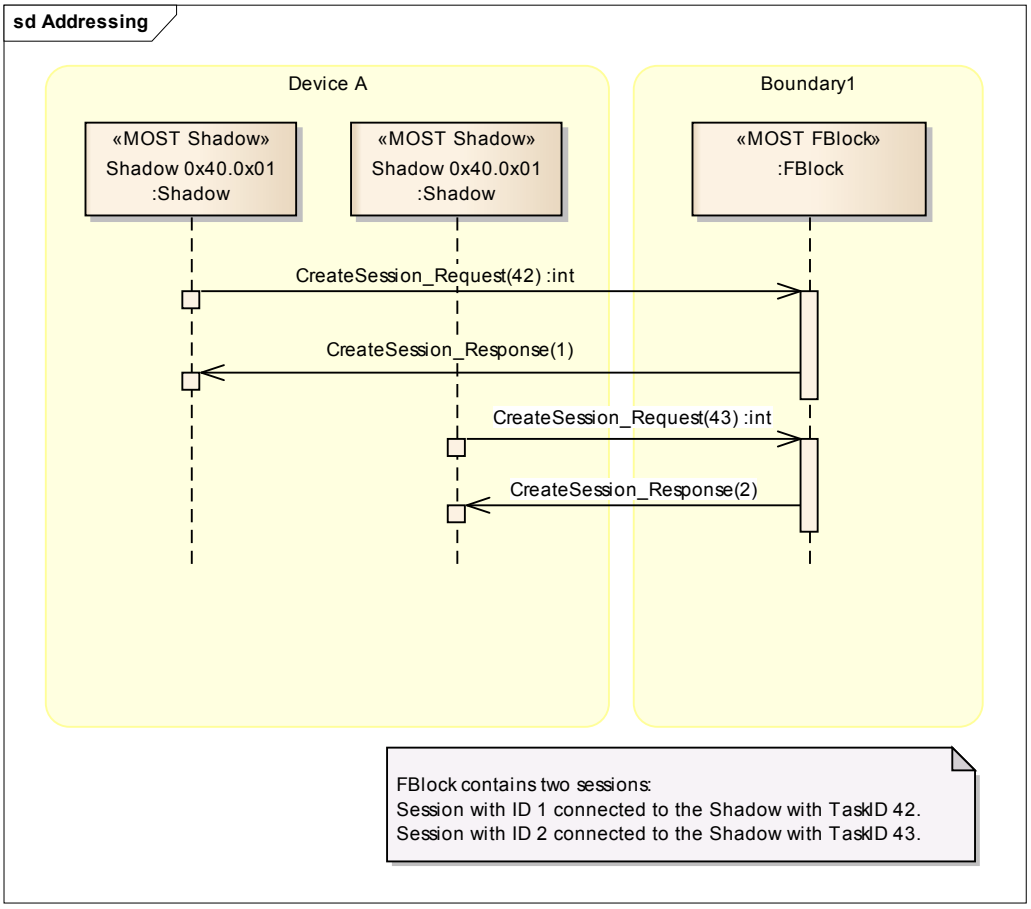

Figure 3-1: Addressing

To make the whole system backward compatible, all functions that contain Session ID and Client ID will be inside the Proprietary (System Specific) Function ID range 0xC00 – 0xEFF. The range from 0xC00 – 0xCFF will be reserved for housekeeping (functions that manage sessions).

SessionID could be used by MOST middleware to find the correct Shadow for report OPTypes. To be able to do that, the middleware needs to store SessionID together with the client context[1] of the connected client.

SessionID will be generated on the FBlock (server) side. It needs to be ensured that SessionID is unique for each registered Shadow. It is not allowed to interpret SessionID. It shall be used as handle.

[1] This depends on the Operating System. This could be a file handle, a socket handle, or a connection ID.

3.1.1 Instance ID handling

Only static Instance IDs will be supported inside the Recommended Communication Profile. Dynamically calculated instance IDs are not supported. This means that the System Integrator chooses Instance IDs in static way.

There are some special Instance ID values (wildcards) that can be used when addressing FBlocks (servers). Please note that this is used only when a Shadow addresses an FBlock and not vice versa. The following are wildcard cases:

Wildcard	Description
0x00	Message will be sent to any FBlock instance. The Recommended Communication Profile recommends always using the lowest available instance ID.
0xFF	Message will be sent to all instances of the addressed FBlock.

Table 3-3: Wildcard addressing

All other rules and exceptions apply from MOST Specification [1].

3.1.2 Logical Device ID

The recommendation of the Recommended Communication Profile is that the Logical Device ID is calculated in dynamic way. This means: based on node position. A Static Logical Device ID is also supported. In this case, the Device ID is statically configured in Network Interface Controller (NIC).

3.2 Operation Types

The Recommended Communication Profile only uses a subset of existing Operation types (OPTypes). The following chapters specify the used OPTypes for Properties and Methods. See also chapter *Compatibility with MOST Specification*.

3.2.1 Operation Types for Properties

Commands	
0x00	Set
0x01	Get
Reports	
0x0C	Status
0x0F	Error

Table 3-4: Operation types for Properties

3.2.2 Operation Types for Methods

Commands	
0x06	StartResultAck
0x07	AbortAck

Commands	
0x08	StartAck
Reports	
0x09	ErrorAck
0x0D	ResultAck

Table 3-5: Operation types for Methods

3.3 Error handling

Errors are indicated through Error (0x0F) and ErrorAck (0x09) operation types. The Error format is completely compatible with the MOST Specification [1]. On the other side, this is a quite restricted definition. A main requirement is the possibility to extend error information but on the other side being compatible with systems which do not support extended error information.

Therefore, the following will be done inside the Recommended Communication Profile. All errors will be derived from MOSTBaseError. MOSTBaseError will contain the errorCode attribute. There is also MOSTExtendedError, which derives from MOSTBaseError and which defines an additional attribute errorInfo (as unstructured stream). This makes it possible to extend the MOSTBaseError class also for some user defined errors. See also the following class diagram, which describes MOST error hierarchy:

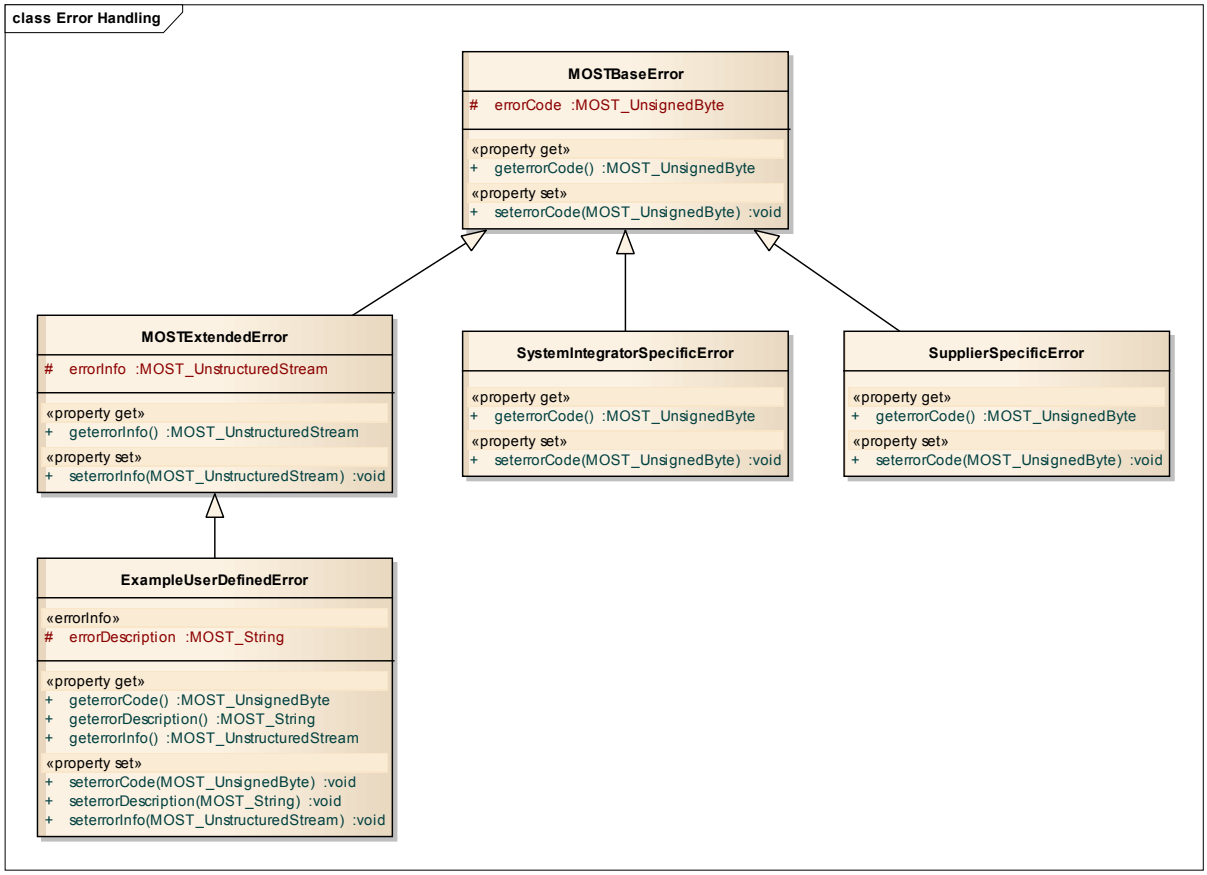

Figure 3-2: Error Handling

For example, if **ExampleUserDefineError** uses **errorDescription** instead of **errorInfo**, it does not matter because **ExampleUserDefinedError** can be cast to MOSTExtendedError at any time. Instead

of **errorInfo**, **errorDescription** will be transferred. Applications which support only **MOSTExtendedError,** read the string as unstructured stream.

3.4 MOST Function Classes

The Recommended Communication Profile will use only the following MOST function classes:

MOST Function Class	Description	Operation Types	Payload
Container	Used for properties. This function class can contain only stream. See also 2.2.1.	Set (0x00)	MOST Stream
		Get (0x01)	
		Status (0x0C)	MOST Stream
Trigger Method	Used for Methods. Contains optionally only Session ID.	StartAck (0x06)	[1]SessionID
		ErrorAck (0x09)	SessionID, MOST Extended Error
		ResultAck (0x0D)	SessionID
Sequence Method	Used for Methods.	StartResultAck (0x06)	SessionID, [TransactionID], MOST Stream
		AbortAck (0x07)	SessionID, [TransactionID], MOST Stream
		ErrorAck (0x09)	SessionID, [TransactionID], MOST Extended Error
		ResultAck (0x0D)	SessionID, [TransactionID], MOST Stream

Table 3-6: Recommended Communication Profile Function Classes

Please note that parameters marked with [] brackets are optional (this depends on the concrete communication pattern).

Some of the mechanisms described in section 3.1 use additional parameters inside MOST payload:

a) SessionID – Addendum to the concept of SenderHandle.

b) ClientID – Replaces SenderHandle (just another name for it) in additional functions which solve Addressing problem.

c) TransactionID – Added additionally to be able to distinguish between different Methods which run in parallel.

All mentioned mechanisms are added only to Methods. That means, only MOST Methods are connection oriented and have the ability to differentiate between different transactions in parallel.

[1] In some communication patterns sessions are not used. In this case Dummy_Sender Handle will be used. For more details please reference chapter 3.6.

3.5 Notification – Push Model

The notification mechanism as defined inside MOST Specification 3.0 [1], has the deficiency that notification is not closed. That means that a call to the Notification.Set() function is not closed with Notification.Status(). Instead, it is closed with the concrete status of properties which were notified. For example:

1) Notification.Set(PropertyA)
2) PropertyA.Status()

To be sure that Notification is closed, the Shadow should trigger the NotificationCheck function and check explicitly if notification worked. For example:

1) Notification.Set(PropertyA)
2) NotificationCheck.Get(PropertyA)
3) NotificationCheck.Status(PropertyA)
4) PropertyA.Status()

Please note that the order of PropertyA and NotificationCheck status is not guaranteed.

The existing notification mechanism as defined in the MOST Specification [1] will be kept without any changes. Notification will still stay connectionless and will not be able to distinguish between different shadows in one device. The exception is the concept of the implicit notification. Implicit notifications **shall not be used** in Recommended Communication Profile systems.

Please take care of the following critical issue when using the notification mechanism.
Example: There are two Shadows in one device. The FBlock resides in a separate device. One of the Shadows clears notification. In this case, the FBlock does not distinguish between the two shadows and clears notification for both of them. When this is done both shadows will be disconnected from the FBlock. This is probably not the intention.

3.6 Communication Patterns

This section will describe communication patterns that will be supported by the Recommended Communication Profile and also concrete Recommended Communication Profile mechanisms which will be used to implement one or the other pattern.

The following communication patterns will be supported on the higher level:

a) Command – Request without Response.

b) Command with Acknowledge – Request with Boolean Response (success, fail). It supports, optionally, exceptions (ErrorAck for example).

c) Bounded Request Response (ID restricted) – Request and Response message contain data. Request and Response have unique ID in scope of the current session. For example:

Request $_{SessionID,\ TransactionID}$ *(Input Data) -> Response* $_{SessionID,\ TransactionID}$ *(Output Data)*

   a. With time-out handling

   Request $_{SessionID,\ TransactionID}$ *(Input Data) -> Error* $_{SessionID,\ TransactionID}$ *(Timeout)*

d) Unbounded Request Response (ID unrestricted) – Similar to previous Request Response, only without support for TransactionID and TimeOut. For example:

Request $_{SessionID}$ *(Data) -> Response* $_{SessionID,}$ *(Data)*

The following example is for the session creation function:

Request $_{ClientID}$ *(Data) -> Response* $_{SessionID,}$ *(Data)*

e) Property Get (Pull Model) – Requests property without any input data.

f) Property Subscription (Push Model) – We have two types of requests here: Subscribe and Unsubscribe.

g) Event Subscription – Very similar to (f) but has no current state and it is used to trigger an event from the server side.

The following common terms will be used in subsequent sections:

- Service – This represents a service which implements functionality. In MOST terms this is an FBlock.
- Client – This is a client which communicates with a service. In MOST terms this is a Shadow.
- Session – Represents a semi-permanent interactive information interchange. Also called dialogue.
- Transaction – Identifies a Request/Response pair which builds a closed transaction unit (used only in combination with Bounded Request Response).

All communication patterns that are session oriented will use the MOST function ID range from 0xC00 to 0xEFF. The sub function range from 0xC00 to 0xCFF will be reserved for housekeeping (functions that manage sessions or some other functions which could be added in the future).

Some of the communication patterns need to contain a SenderHandle in payload just to keep compatibility with the MOST Specification [1]. Operation types Start or StartResult could be used in this case, but these operation types are deprecated. This SenderHandle is not interpreted and content does not matter. This is called Dummy_SenderHandle. Dummy_SenderHandle should be always set to value 0xFFFF. Please note that SessionID could contain the same value. As noted before, this value should not be interpreted.

3.6.1 Timeout handling

In the MOST Specification [1], timeout is measured on the server side. Timeout is in this case used for periodic notification if a Method still runs. This is indicated by sending back OPType Processing or ProcessingAck. This concept is not used inside the Recommended Communication Profile. See also section *3.2 Operation Types*.

Timeout measurement is done only on the Shadow side. Timeout is defined optionally for Methods where this is appropriate (**Timeout handling is not done for properties**). Per default, no timeout measurement will be done on the Shadow side (unless otherwise defined). In the case that a timeout is defined for a Method, it is mandatory to implement timeout handling on the Shadow side. This could be defined inside, for example, the MOST Function Catalog.

Timeout handling must be used only in conjunction with the *3.6.4 Bounded Request Response (ID restricted)* communication pattern (a pattern which uses transaction).

Timeout measurement starts at the moment when a Method is triggered. In the case that timeout elapses, the transaction will be terminated and will be considered as timed out. In the case that a response arrives for an already timed out transaction, it will be ignored (additional assertions could be performed inside the system to detect slow reaction of the FBlock/Server side).

3.6.2 Command

Commands are used to trigger actions on the service side. There is no response when using the command pattern. A command is executed immediately after reception on the service side. Commands are not connected to any sessions or transactions. **A command does not contain any payload except Dummy_SenderHandle (for more details see 3.6)**.

Function class Trigger will be used in this case:

MOST Function Class	Description	Operation Types	Payload
Trigger	StartAck represents a command.	StartAck (0x06)	Dummy_SenderHandle

Table 3-7: Command

3.6.3 Command with Acknowledge

Command with Acknowledge is used to trigger actions on the service side. The service returns a response which contains one of the following two possible values:

a) Success – Operation succeeded

b) Failed – Operation failed

MOST Function Class	Description	Operation Types	Payload
SequenceMethod		StartResultAck (0x06)	SessionID, TransactionID
		ErrorAck (0x09)	SessionID, TransactionID, MOST Extended Error
		ResultAck (0x0D)	SessionID, TransactionID, Status

Table 3-8: Command with Acknowledge

Parameter Name	Data Type	Parameter Description
MOST Extended Error	MOST Stream	Stream contains ErrorCode and ErrorInformation.
Status	MOST Boolean	TRUE for success otherwise FALSE.

Table 3-9: Parameters for Command with Acknowledge

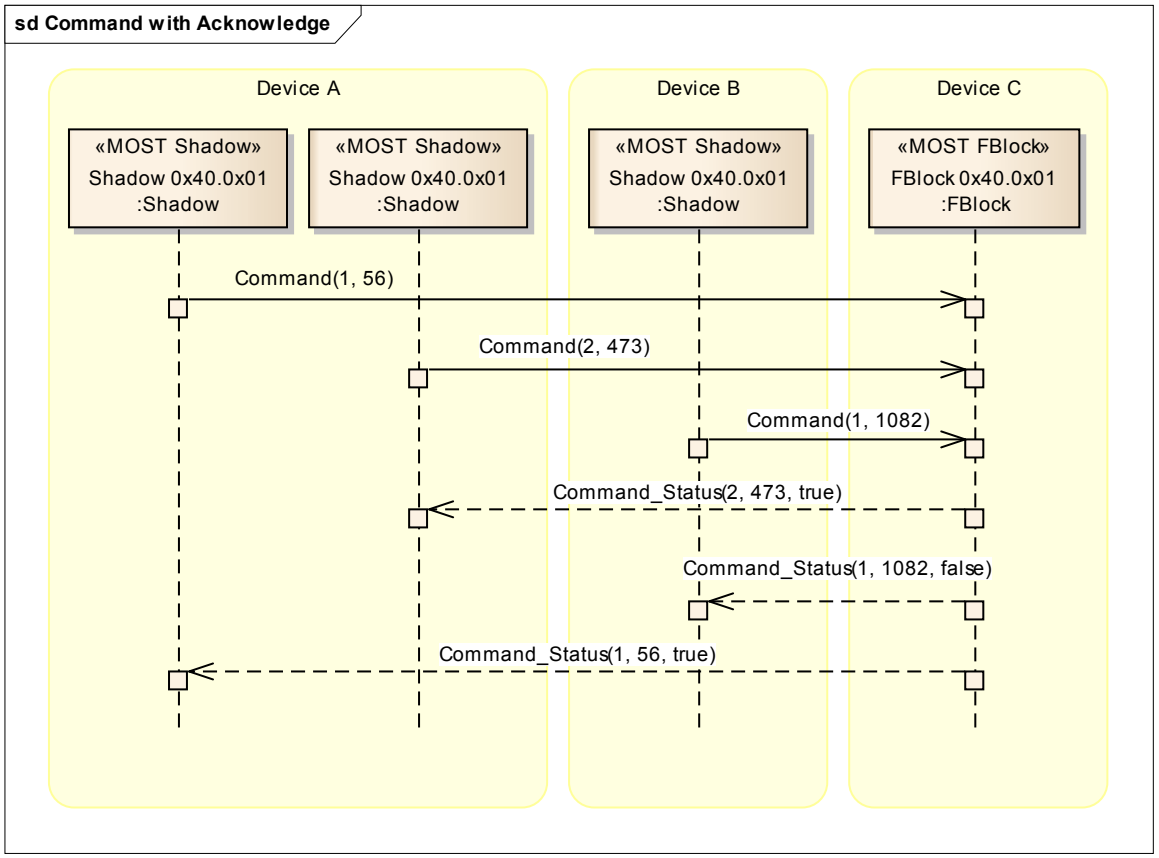

Figure 3-3: Command with Acknowledge

3.6.4 Bounded Request Response (ID restricted)

The Bounded Request Response (ID restricted) is a request response pattern. It is attached to one session between **Shadow** and **FBlock**. Additionally, each Request-Response pair is handled as separate **Transaction**. **Transaction IDs** are created on client side (Shadow). The server can use the Transaction ID to distinguish between the transactions which run inside one session in parallel.

MOST Function Class	Description	Operation Types	Possible Payload
Sequence Method	Used for Methods.	StartResultAck (0x06)	SessionID, [TransactionID], Input data MOST Stream
		AbortAck (0x07)	SessionID, [TransactionID], MOST Stream
		ErrorAck (0x09)	SessionID, [TransactionID], MOST Extended Error
		ResultAck (0x0D)	SessionID, [TransactionID], Output data, MOST Stream

Table 3-10: Bounded Request Response (ID restricted)

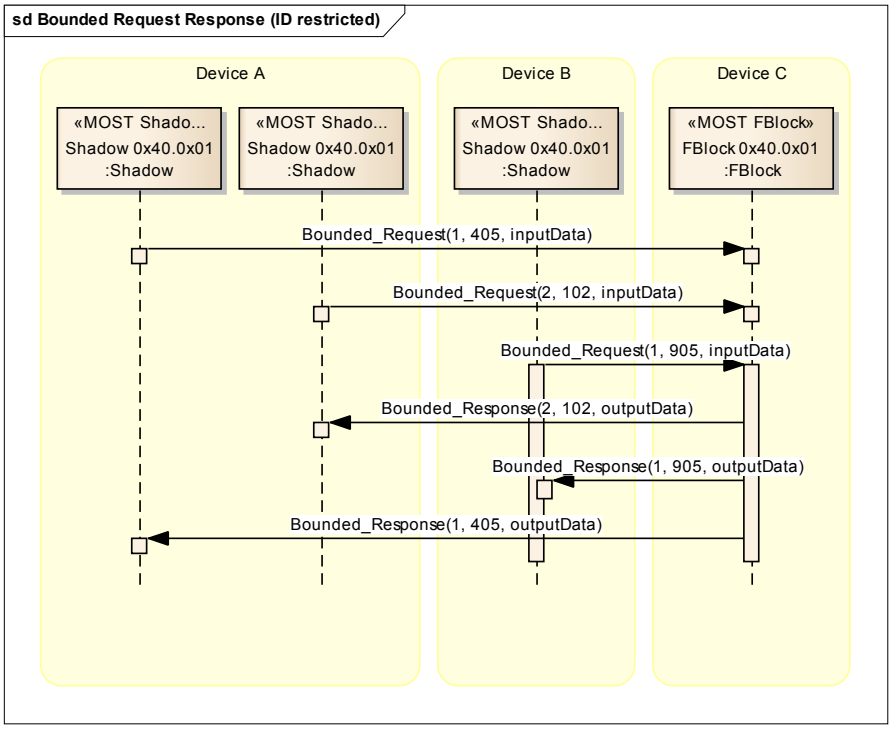

Figure 3-4: Bounded Request Response

3.6.5 Unbounded Request Response (ID unrestricted)

The Unbounded Request Response (ID unrestricted) anticipates that a request is performed and waits for the result. And whatever response is received, it is accepted whenever it is produced, no matter if it is based on the previous identical request or the last request.

An unbounded request does not require a session to be opened. When an unbounded request is used, it is not possible to establish a correlation between a specific Request and Response. But this simplifies the code that supports this pattern, so the implementer does not need to take care of request timeout, and the order of incoming and outgoing messages. It is useful for the data that is not changing in the time, or users do not plan to ask to make the request with different parameters at the same time.

Similar to the command, the unbounded request response will contain a Dummy_SenderHandle. The reason is compatibility with the MOST Specification [1]. The content of this SenderHandle will not be interpreted.

MOST Function Class	Description	Operation Types	Possible Payload
Sequence Method	Used for Methods.	StartResultAck (0x06)	Dummy_SenderHandle, Input data
		ErrorAck (0x09)	Dummy_SenderHandle, MOST Extended Error
		ResultAck (0x0D)	Dummy_SenderHandle, Output data

Table 3-11: Unbounded Request Response (ID unrestricted)

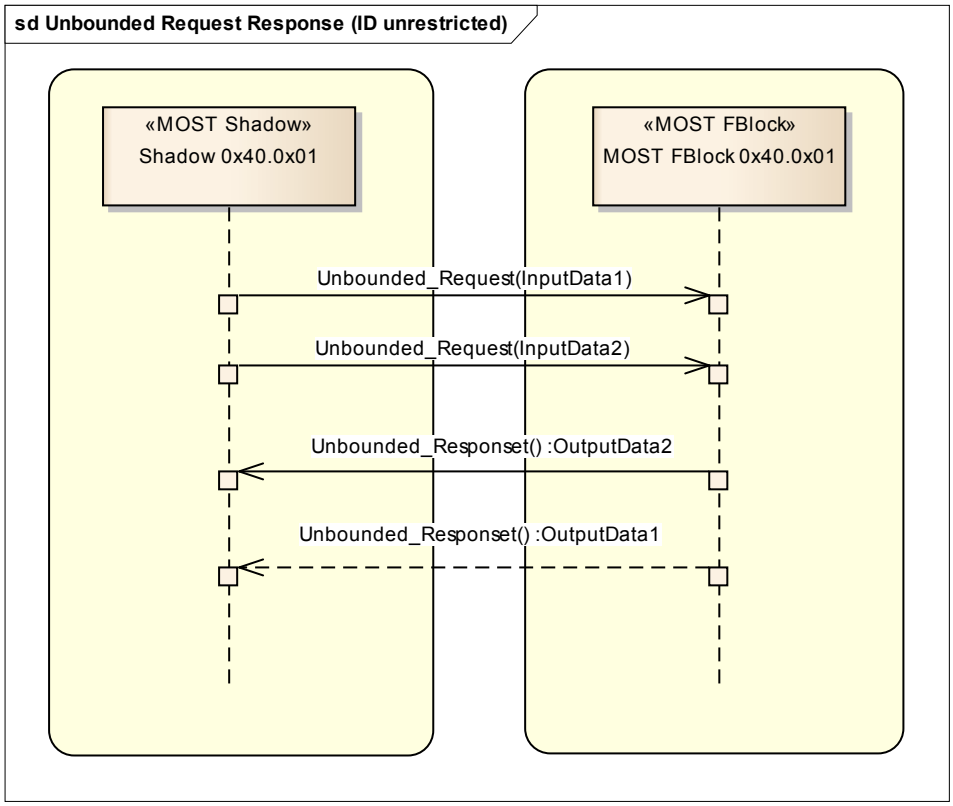

Figure 3-5: Unbounded Request Response

3.6.6 Property Get (Pull Model)

A Property has a current value and does not depend on an incoming parameter that belongs to a request. So the pull model assumes a Get request [1]**without any input parameter** and a Transaction ID to trigger the transmission of a status message.

MOST Function Class	Description	Operation Types	Possible Payload
Container	Used for properties. This function class can contain only stream. See also 2.2.1.	Set (0x00)	MOST Stream
		Get (0x01)	
		Status (0x0C)	MOST Stream

Table 3-12: Property Get (Pull Model)

Please note that Properties are connectionless and therefore there is no guarantee that a Property will be changed by calling OPType Set (message could get lost). If in some use cases this needs to be guaranteed, please use the Bounded Request Response pattern instead. In the case of Bounded

[1] In the case that a parameter is needed, property is not the right choice for this. **It is also explicitely forbidden in this specification.** In this case, other communication patterns like Request Response are more appropriate.

Request Response, the Shadow always receives confirmation that an action is executed. If no response is received, and this is critical, optionally, timeout handling can be activated for this method.

In some cases, confirmation is not needed. For example, if a Property is cyclically set every 100ms.

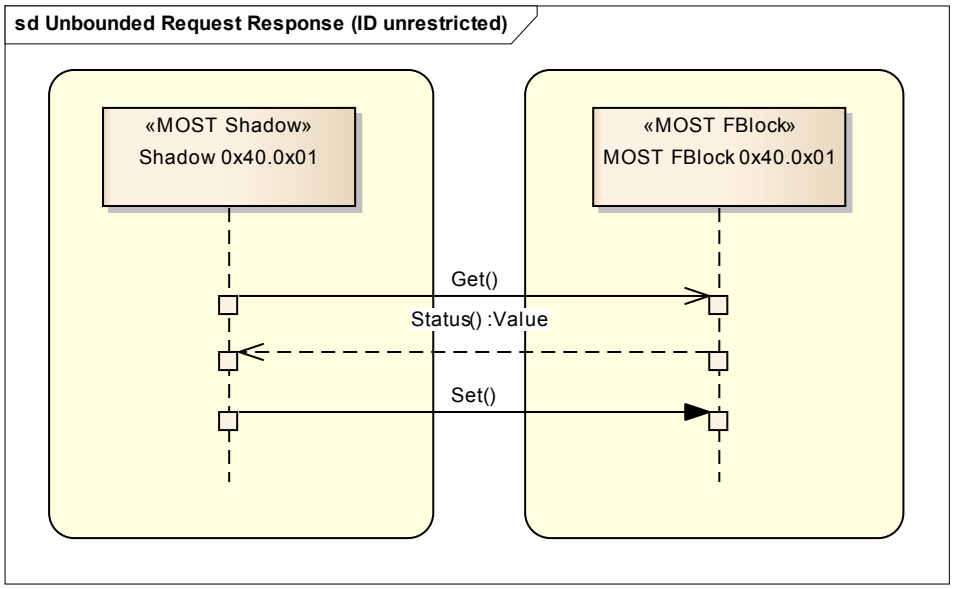

Figure 3-6: Pull Model

3.6.7 Property Subscription (Push Model)

In the push model, the value for the Property is transmitted by the FBlock, whenever notification is necessary. The reason could be a property change or timer-caused notification. The push model can be combined with the pull model; this means that the Property value can be requested by Get as well.

MOST Function Class	Description	Operation Types	Possible Payload
Container	Used for properties. This function class can contain only stream. See also 2.2.1.	Set (0x00)	MOST Stream
		Get (0x01)	
		Status (0x0C)	

Table 3-13: Property Subscription (Push Model)

The Notification and Notification Check functions are also needed. See the MOST Specification [1] for more details about these functions.

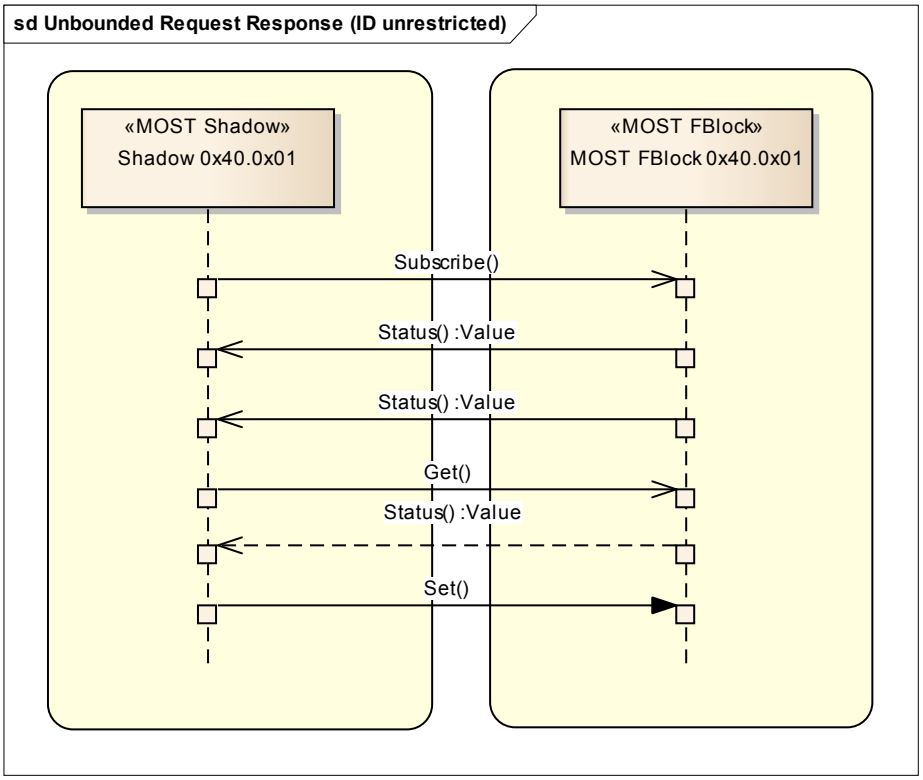

Figure 3-7: Push Model

3.6.8 Event Subscription

An event can be interpreted as a Property which has no current value and status, but the set of events in time has meaning. There are no Set or Get operations for such properties but only Subscribe and Status.

MOST Function Class	Description	Operation Types	Possible Payload
Container	Used for properties. This function class can contain only stream. See also 2.2.1.	Status (0x0C)	

Table 3-14: Event Subscription

The Notification and Notification Check functions are also needed. See the MOST Specification [1] for more detail about these functions.

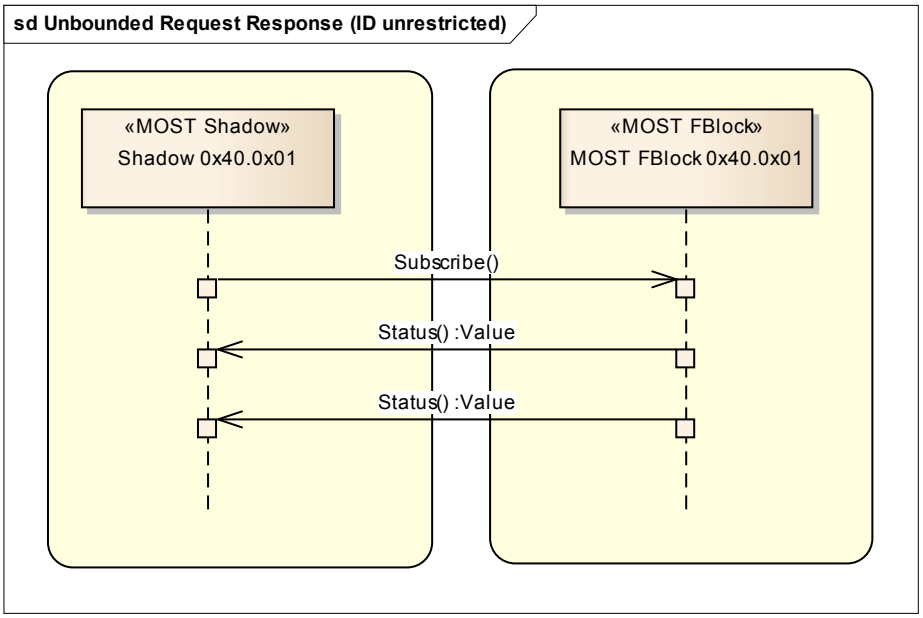

Figure 3-8: Event

4 Functions organization

4.1 Interfaces

Each Recommended Communication Profile FBlock implements its functionality by a set of incoming and outgoing messages. These messages could be considered as protocol implemented by an FBlock. Each message has an internal number unique within this FBlock, called function ID.

Some of the messages could be the same for different FBlocks, as Play, Pause, and Stop, so we could define the sub protocol of the FBlock and put it out as separate contract that could be reused in different FBlock. Such contracts can be interpreted as programming interfaces.

Each such interface consists of Methods and Properties, restricted by communication patterns described above. The input parameters are transmitted by outgoing messages, the return value if present by incoming messages for the FBlock's client.

So Play, Pause, Stop functionality could be represented by interface in UML model.

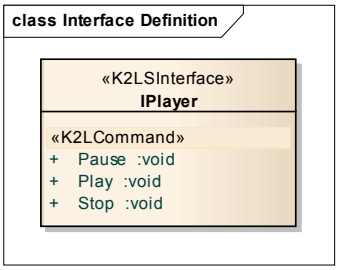

Figure 4-1: Interface

"K2LCommand" corresponds to Command communication pattern described in 3.6.2

4.2 Interfaces resolution

The interface is independent of an FBlock and it defines how the messages will obtain their function IDs in a particular FBlock.

It is possible to say that each interface will implement a predefined set of function ID in each FBlocks. Pause can take function ID 0x800, Play 0x801, and Stop 0x802 but it will be required to reserve some ranges for dedicated function IDs and support its consistency, so that nobody can define interfaces with the same IDs.

Another approach proposes a virtual function ID, like a virtual table in C++, where each method (message) in the interface has only a relative function ID inside the interface, defined by the message order. The real function ID is assigned to a method when it is implemented by a particular FBlock.

If an FBlock implements two interfaces, each method shall be sequential in one group (without any gap).

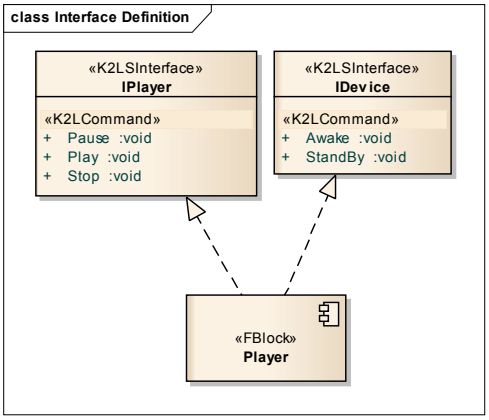

Figure 4-2: FBlock implements interfaces

Interface	Method	Method number	Function ID
IPlayer	Pause	1	0xD10
	Play	2	0xD11
	Stop	3	0xD12
IDevice	Awake	1	0xD13
	Standby	2	0xD14

When modeling interfaces with a UML design tool, the number can be assigned to methods by tags to avoid relying on sorting rules of the methods inside UML.

4.2.1 Static binding

When production code is generated from the model, the FBlock will obtain all function IDs for all messages inside its protocol; apart from that, these messages will be known to the client proxy, called Shadow in MOST. Such an approach does not require any additional interface resolution in the link of the Shadow and FBlock.

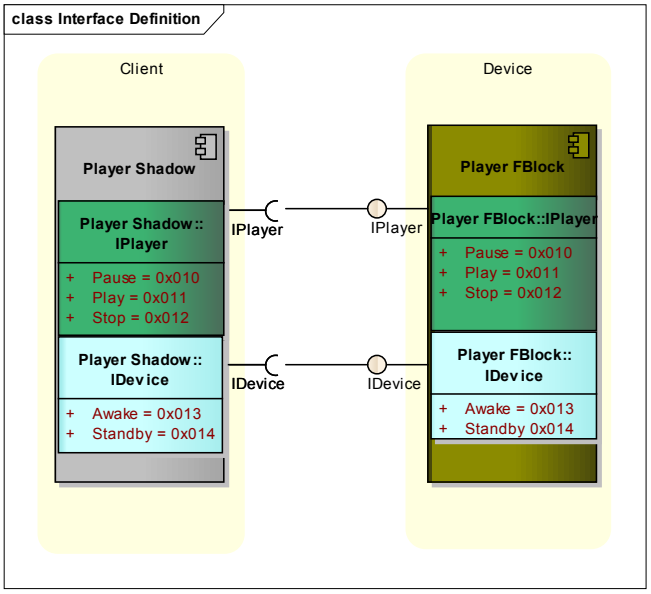

Figure 4-3: Shadow and FBlock have the same functionID table

4.2.2 Dynamic binding

If the number of function ID of the particular interface that is implemented by an FBlock is not known at the moment of code compilation, it is possible to resolve the interface dynamically.

In such a case, the Shadow shall know only the Device ID and FBlock ID. It can use a special set of defined functions that shall be implemented by the FBlock if it supports dynamic resolution.

Knowing the relative number of each method in the interface, the shadow can ask the FBlock of the start number of the particular interface and create function IDs dynamically for it.

That allows working with different FBlocks via a well-known interface without creating the precompiled Shadow for each.

The following table defines resolution functions:

Function	ID	Parameters	Result
IsA	0xC03	String name of the interface. The proposed name could be Company.Product.Interface	True or False
Resolve	0xC04	String name of the interface. The proposed name could be Company.Product.Interface	The list of pairs: *<Parent Interface> – <Start function ID>* and *<Interface> – <Start function ID>* *See the Versioning Support

Table 4-1: Dynamic binding

4.3 Inheritance

Determining a subset of FBlock functionality as an interface allows applying the powerful OOD feature – inheritance.

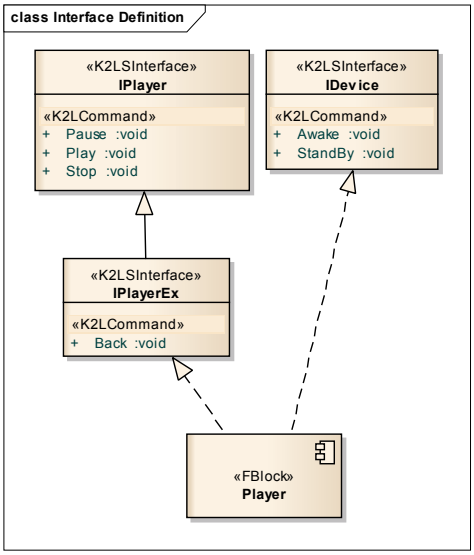

Figure 4-4: Interface inheritance

Virtual inheritance will not be supported. In case of parent interface resolution, implemented in more than one child interface, the function ID for the first implementer will be returned.

4.4 Versioning support

Inheritance is a private case of versioning, where an FBlock can be updated by either extending one of its interfaces or adding a new interface. In this case, backward compatibility at the level of a set of functions shall be supported to allow for old shadows to communicate with the new version of an FBlock.

For backward compatibility, the protocol is not allowed to change the order of existent messages (functions inside interfaces), but it may be extended. That means that all new added members of each added interface shall achieve function ID numbers greater than all from the previous version.

The following rules shall be applied to interfaces:

1. It is not possible to extend the existing interface or delete any method from it.
2. To add new functionality, the new interface shall inherit the older version of the interface and new functionality **is contained only in the new interface**.
3. The FBlock model element in a UML design tool shall keep the sequence of implemented interfaces (this could be done by using tags). .

The extended interface part will be placed in the function table above previously implemented interfaces.

This will allow the old Shadow to work with a new FBlock using the same function IDs. Dynamic resolution for IPlayerEx will return start functionID for its parent IPlayer and functionID for IPlayerEx part.

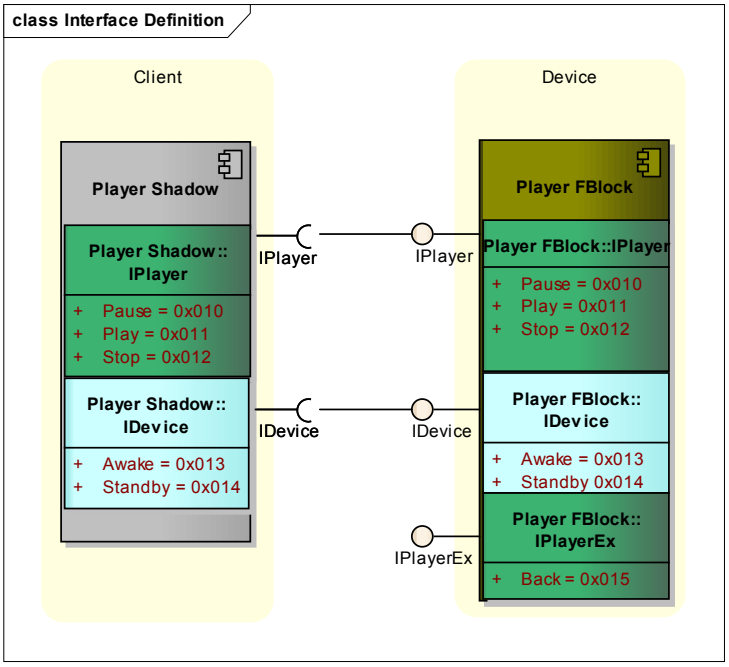

Figure 4-5: FBlock of new version extends its table of functionIDs

4.5 Advanced Version Support Issues

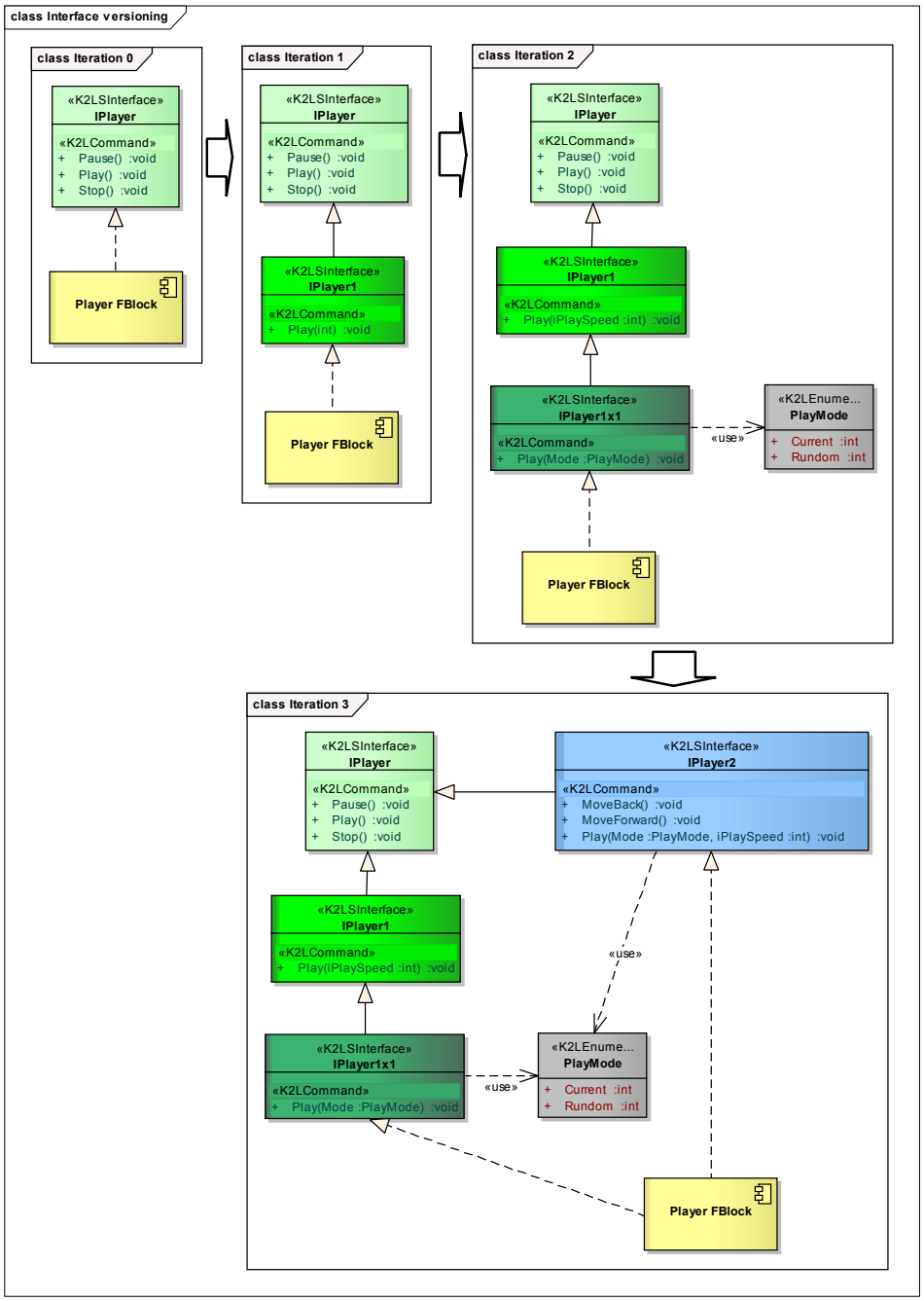

Figure 4-6: Interface evolution

The picture above shows the complex design evolution from IPlayer through IPlayer1 to IPlayer1x1 as one branch and as IPlayer2 as an orthogonal branch of the IPlayer. IPlayer2 does not inherit any ads made in IPlayer1... series. This scenario could apply if one developer updates the trunk from IPlayer to IPlayer 2 and another provides bug fixing and minor interface changes at the same time.

The major issue would be to support backward compatibility for the released devices.

If the clients released for IPlayer1x1 are deployed and the next major release is planned to support IPlayer2, the released server (FBlock) shall support both IPlayer1x1 and IPlayer2. In terms of protocol that means that the protocol for the new FBlock shall contain messages for IPlayer, IPlayer1, IPlayer1x1 and IPlayer2. And the IDs (FunctionIDs) of those messages should remain the same as for

previous releases of the same FBlock. This means that the protocol is combined in the same order as it was combined for previous releases and all additions shall be placed at the end.

Here is the evolution in details:

There is the table of the protocol for the initial FBlock that supports only IPlayer. The table shows the relative message number, assigned to a command (method) in the interface and provides the absolute number inside this particular FBlock.

Function (message)	Relative number	Absolute number (Function ID) in FBlock
IPlayer		
Pause	0x1	0xD10
Play	0x2	0xD12
Stop	0x3	0xD13

Table 4-2: IPlayer implementation

It is assumed that the FBlock for IPlay is released and clients were written to it. And after that, the development is started for the next generation of the FBlock, for planned version IPlayer2. But in the meantime, the support team decided to fix a design error in the IPlayer interface and extend the method Play with parameter for play speed.

They decided to release this interface before IPlayer2

Function (message)	Relative number	Absolute number (Function ID) in FBlock
IPlayer		
Pause	0x1	0xD10
Play	0x2	0xD12
Stop	0x3	0xD13
IPlayer1		
Play(int iPlaySpeed)	0x4	0xD14

Table 4-3: IPlayer1 implementation

In terms of interfaces, IPlayer1.Play(int iPlaySpeed) overwrites IPlayer.Play() but in terms of protocol, these are different messages, that shall be supported both in the device for compatibility.

All old clients, which contain table 1 will work with new FBlocks, which contain table 2.

Furthermore, before the release of IPlayer2, the support team wanted to extend the IPlayer1 interface with another overwritten method.

Function (message)	Relative number	Absolute number (Function ID) in FBlock
IPlayer		
Pause	0x1	0xD10
Play	0x2	0xD12
Stop	0x3	0xD13
IPlayer1		

Function (message)	Relative number	Absolute number (Function ID) in FBlock
Play(int iPlaySpeed)	0x4	0xD14
IPlayer1x1		
Play(PlayMode mode)	0x5	0xD15

Table 4-4: IPlayer1x1 implementation

The new method/message is added to the end of the protocol.

If the FBlock released, this order of methods/messages shall be kept during all of its lifecycle if the compatibility is anticipated.

At the end, the development department wants to release IPlayer2, but it is based on IPlayer and has not IPlayer1 and IPlayer1x1, because it is redesigned.However, it is still required that clients which were released for IPlayer1 an IPlayer1x1 can work with new FBlock. It means that the new FBlock shall implement all of the interfaces and its protocol should be the following:

Function (message)	Relative number	Absolute number (Function ID) in FBlock
IPlayer		
Pause	0x1	0XD10
Play	0x2	0XD12
Stop	0x3	0xD13
IPlayer1		
Play(int iPlaySpeed)	0x4	0xD14
IPlayer1x1		
Play(PlayMode mode)	0x5	0xD15
IPlayer2		
MoveBack	(inherits IPlayer) 0x4	0xD16
MoveForward	0x5	0xD17
Play(int iPlaySpeed, PlayMode mode)	0x6	0xD18

Table 4-5: IPlayer2 implementation

FBlock compatibility rule: *The absolute number of messages (Function IDs) for an FBlock's protocol (the full bunch of the interfaces it implements) shall be kept among the versions, where new messages shall be added to the end of the protocol.*

Any shadow that supports only IPlayer2, will skip the IPlayer1 and IPlayer1x1. Message numbers will be the same on Shadow and FBlock side.

It shall be possible to select a subset of interfaces inside a particular FBlock that will be supported by the Shadow. This is typicaly done to reduce Shadow size. For example, an FBlock implements multiple interfaces but a specific Shadow needs only one of these interfaces. This is usually needed when generating runnable code from a UML design model .

The table of such a Shadow will be the following:

Function (message)	Relative number	Absolute number (Function Id) in FBlock
IPlayer		
Pause	0x1	0xD10
Play	0x2	0xD12
Stop	0x3	0xD13
IPlayer2		
MoveBack	(inherits IPlayer) 0x4	0xD16
MoveForward	0x5	0xD17
Play(int iPlaySpeed, PlayMode mode)	0x6	0xD18

Table 4-6: Message table for Shadows which support only IPlayer2

4.6 Dynamic Interface Resolution

Dynamic resolution means that a client during compilation does not know with which particular FBlocks it will communicate with. What is known, is the common interface which defines the set of methods/messages and rules for serialization of its parameters. See section 4.2.2 for more details. The set of messages means that the client knows only relative message number from the beginning of the interface and does not know the real Function ID of the particular device.

To obtain the real Function ID, the client resolves the known interface of an FBlock with known address and by using dynamic resolution mechanisms (see Dynamic binding). The FBlock provides the starting Message ID for this interface and all of its ancestors (see previous chapter on inheritance). The client calculates the Function ID for this block dynamically.

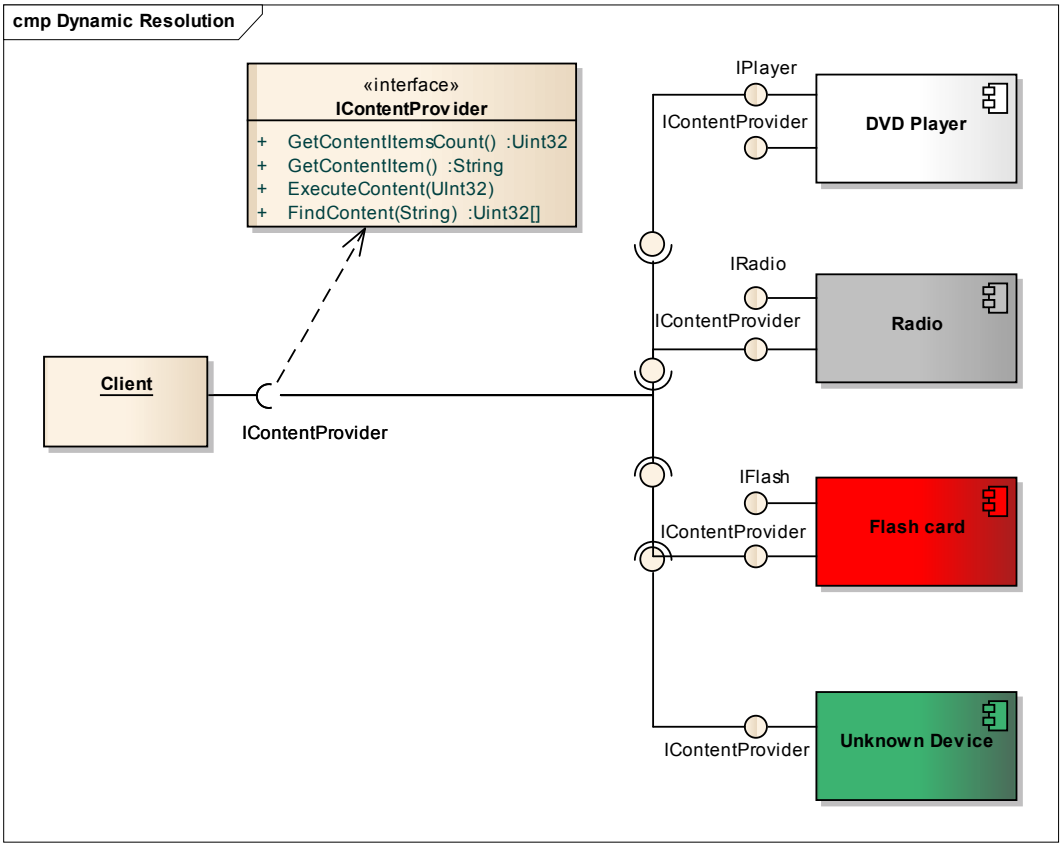

Figure 4-7: Dynamic resolution: Component Model

For instance, there are is a concrete FBlock, which implements the common IContentProvider interface that is used for querying a media library.

The interface could be represented as

Function (message)	Relative number	Description
UInt32 GetContentItemsCount()	0x1	Return amount of available items
String GetContentItem(UInt32)	0x2	Return context description
EexecuteContext(UInt32)	0x3	Perform device-specific action for context
UInt32[] FindContents(String)	0x4	Find items suting the filter

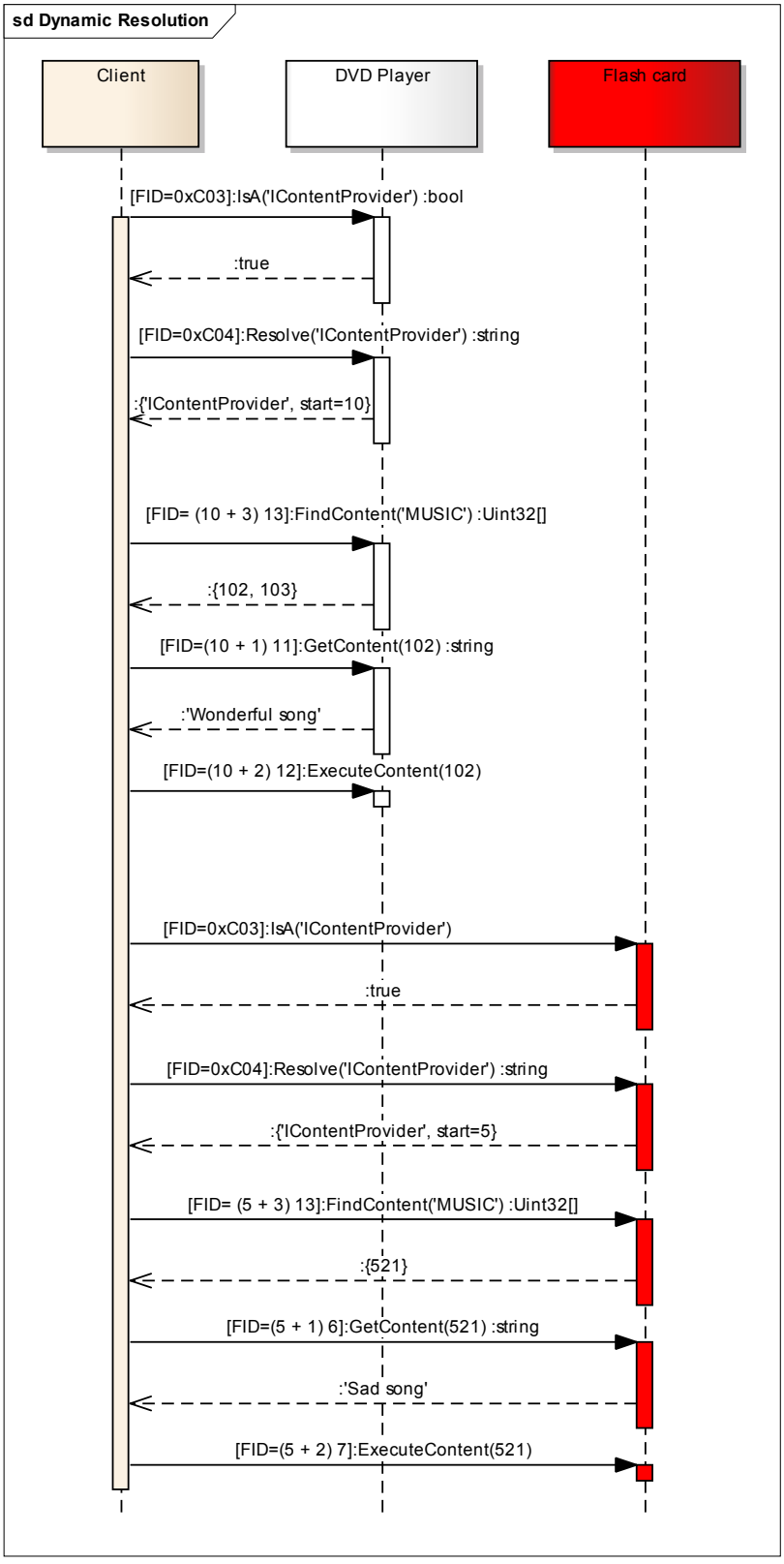

Figure 4-8: Dynamic resolution: Sequence chart

The diagram shows that the client asks for each available device if it implements the IContentProvider interface. It dynamically detects the Function ID on the device. It allows executing known functions and using known serialization code.

4.7 Function FktIDs

It is mandatory that an FBlock implements function FktIDs. Nevertheless, it is important to note that this function will not be used to make any assumptions about implemented interfaces on the Shadow side. That means, the Shadow is **not allowed to make any assumptions based on the content of FktIDs.Status.** Instead, the Shadow should use the previously mentioned Interface Resolution approaches described in Interfaces resolution.

5 MOST High Protocol

The MOST High Protocol (MHP) supports data transfers up to 65535 (0xFFFF) bytes in one chunk. MHP supports also transfers that are bigger than 65535 bytes. In this case, one message is transferred in multiple chunks. Because it is impractical to allocate memory for messages bigger than 64KBytes, structured data is limited to only one chunk meaning 65535 bytes. Only **Unstructured Streams** are supported for transfers bigger than 65535 bytes. Additionally, practice is that data is read in blocks. For example, if the use case is reading a 5 MByte file and sending it to the other MOST device, it can be accomplished by reading blocks of 64KBytes and sending each block until the complete file is transferred. It is impractical to allocate memory for 5MBytes and read the file from disk into a 5MByte region before sending.

From the perspective of developer, the the Recommended Communication Profile environment could provide the stream of bytes, which will receive long bunch of data. The stream shall take care of MOST High Protocol manipulation, in order to hold on the transmission if the incoming buffer is close to being overrun.

6 Compatibility with MOST Specification

The Recommended Communication Profile uses only one part of the OPTypes defined inside MOST Specification [1]. It is possible that some FBlocks use some of the OPTypes not specified in the Recommended Communication Profile (for example SetGet). To be compatible with MOST Specification 3.0 [1], deviations will be allowed for some FBlocks. Deviations are allowed for following FBlocks:

1) NetBlock
2) ConnectionMaster
3) PowerMaster
4) EnhancedTestability
5) NetworkMaster

This is nevertheless no problem because these FBlocks are implemented inside the Middleware and not Customer Applications.

Appendix B: Index of Figures

Appendix C: Index of Tables

Notes: