MOST

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

Multimedia and Control Networking Technology

MOST MSC Cookbook
Rev 1.2
04/2005

Version 1.2-00

Legal Notice

COPYRIGHT

LICENSE DISCLAIMER

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

CONTENT AND LIABILITY DISCLAIMER

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

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

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

FEEDBACK INFORMATION

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

TRADEMARKS

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

SUPPORT AND FURTHER INFORMATION

For more information on the MOST technology, please contact:

MOST Cooperation
Administration
Bannwaldallee 48
D-76185 Karlsruhe
Germany

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

Contents

Document References

This document is a Guideline and may therefore refer to any document published by MOST Cooperation or any other documentation listed in the references.

References

[1] **ITU-T Recommendation Z.120** (MSC2000)
Message Sequence Chart (MSC)
Geneva, (11/1999)
[2] **ITU-T Z.120C1** LC-Text: Recommendation
Z.120 Corrigendum 1
(11/2001)
[3] **ITU-T Recommendation Z.100** (SDL)
Specification and Description Language (SDL)
(11/1999)

Definitions

The following table explains abbreviations, and contains definitions for terms used throughout this document.

Term	Description
Component	The term “component" is used for hardware components (devices) as well as software components (applications).
HMI	Human Machine Interface. This is the user interaction part. We use “HMI” instead of the more common “MMI” to distinguish between the user instance and the MMI FunctionBlock (0x10).
HMSC	High Level Message Sequence Chart. HMSCs are used to describe the relations between a number of MSCs.
MMI	Man Machine Interface. Whenever the term “MMI" is used, we intend to address the MMI FunctionBlock (0x10).
MSC	Message Sequence Chart. Commonly used to refer to a single Basic MSC.
MSC-GR	The graphical representation of MSCs.
MSC-PR	The textual representation of MSCs.
MSC standard	The MSC standard is officially called “recommendation Z.120” [1]. To better distinguish between our recommendations (guidelines) and the official “MSC recommendation”, we will use the term “MSC standard” instead of “recommendation Z.120” or “MSC recommendation” in this document.
MPR	The most common file extension for MSC-PR. It is quite common to use the term “mpr file” when talking about the storage format.
Named parameters	An extension to the MSC standard to facilitate use of MSCs in the telematics and in-car infotainment domains.
SDL	Specification and Description Language. In this document, we use a few simple SDL constructs to describe operations on data.
Z.100	SDL is specified in ITU-T recommendation Z.100 [3].
Z.120	MSCs are specified in ITU-T recommendation Z.120.

Document History

Changes MOST MSC Cookbook 1V0-00 to MOST MSC Cookbook 1V2-00

Change Ref.	Section	Changes
-	-	Initial revision
1V1-00	All	Several aspects refined. Inconsistencies resolved.
1V2-00	Examples	Additional chapter for dealing with MOST Catalog types.

1 Introduction

Welcome to the MOST MSC Cookbook!

Message Sequence Charts (MSCs) offer an intuitive description of communication in complex distributed systems. The following document presents an insight into the application of MSCs within the telematics and in-car infotainment domains.

1.1 Motivation

We all share the need for requirements that are more formal than textual descriptions. The realization that MSCs can help to bridge the gap between requirements and automated testing has resulted in the creation of this document, which will hopefully help you to avoid the most common mistakes.

The proven success of MSCs as a tool for the creation of formalized specifications has led to the authors' commitment to make MSCs a central building block of their telematics and infotainment system specification.

One of our main goals is to reuse system specification MSCs for testing. To be able to do that, a number of things need to be considered, and some foresight is required. Many of the recommendations we make throughout this document are based on testability requirements.

The purpose of the MSC Cookbook is to

- promote testability of MSCs
- promote correctness
- promote disambiguation
- help in preventing side effects

1.2 Target Audience

The goal of the MSC Cookbook is to regulate the use and application of MSCs. It shall serve as a guideline for all who get in contact with MSCs during their work.
This document shall serve as the main reference throughout the process of specification and development and addresses all the stakeholders involved, mainly

- Creators of specifications
- Implementers
- Test Engineers

1.3 Overview

The document is subdivided in the following main sections:

- Message Sequence Charts - an Introduction
- Specific adaptation of MSCs for telematics and in-car infotainment
- MSC Notation and Guidelines for MSC Composition

The MSC introduction section illuminates the idea behind Message Sequence Charts. The telematics- and infotainment-specific adaptation section contains a number of recommendations for use in the respective domains. The extensive notation and guidelines part offers a detailed guide to relevant MSC notation and MSC composition. Rather than just explaining MSC constructs in general, we focused on the use in the field of telematics and in-car infotainment, also pointing out a number of pitfalls that we have come across in the past.

1.4 Request for Comments

If any construct that you need for your specification is not described here, please contact the coordinator of Working Group Device Architecture. It is of utmost importance to us that all MSCs are kept consistent.
Also, your suggestions are always very welcome. So if you discover a gap in this cookbook, and you know how to fill it, please share your knowledge!

For questions, please contact the following persons:

Name	E-Mail Address
Mr. Matthias Boll, DaimlerChrysler AG	matthias.boll@daimlerchrysler.com
Ms. Leila Öhman-Denton, Oasis SiliconSystems AB	leila.ohman-denton@oasis.com
Dr. Bernd Sostawa, BMW AG	bernd.sostawa@bmw.de

2 Message Sequence Charts – An Introduction

In the automotive domain, MSCs are mainly used to describe the dynamic behavior of the distributed telematics and in-car infotainment system. Syntax and semantics of Message Sequence Charts (MSCs) have been standardized by the International Telecommunications Union (ITU). The relevant standard is ITU-T Z.120 (MSC2000) [1], including the Corrigendum1 [2].

The fact that a precisely defined grammar exists for MSCs makes them easily interchangeable between tools from different suppliers.

In the past, several graphical notations for component interaction have been suggested. Out of these notations the Message Sequence Charts gained a comparably strong position as an intuitive form of describing asynchronous message exchange between instances in a communications system.

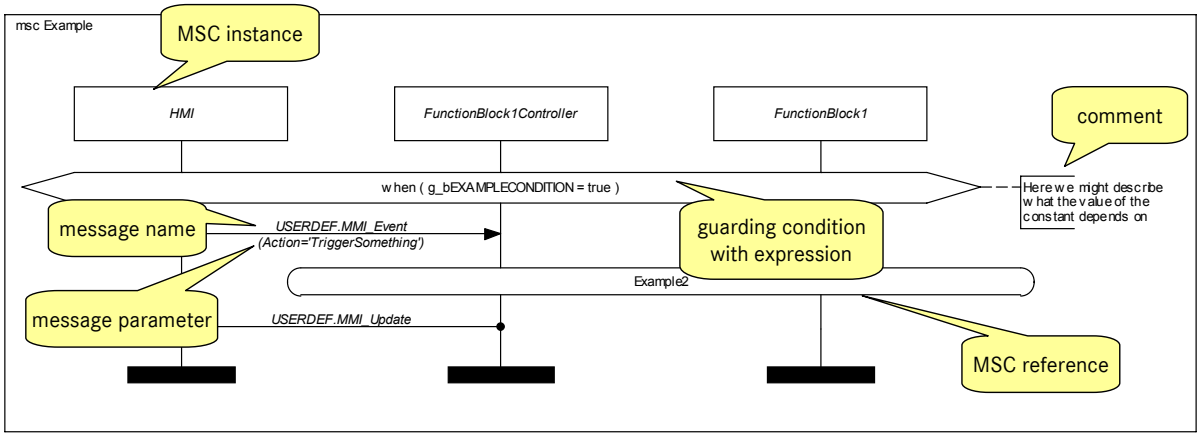
The origin of the semiformal notation lies in the areas of Requirements Engineering, Specification and Design during the development of telecommunications switching systems. MSCs have always been one of several views on an SDL-Model (Specification Description Language). In the meantime MSCs have emancipated from SDL and their standardization process is run by the ITU.

"The purpose of recommending MSC is to provide a trace language for the specification and description of the communication behavior of system components and their environment by means of message interchange. Since in MSCs the communication behavior is presented in a very intuitive and transparent manner, particularly in the graphical representation, the MSC language is easy to learn, use and interpret. In connection with other languages it can be used to support methodologies for system specification, design, simulation, testing, and documentation."

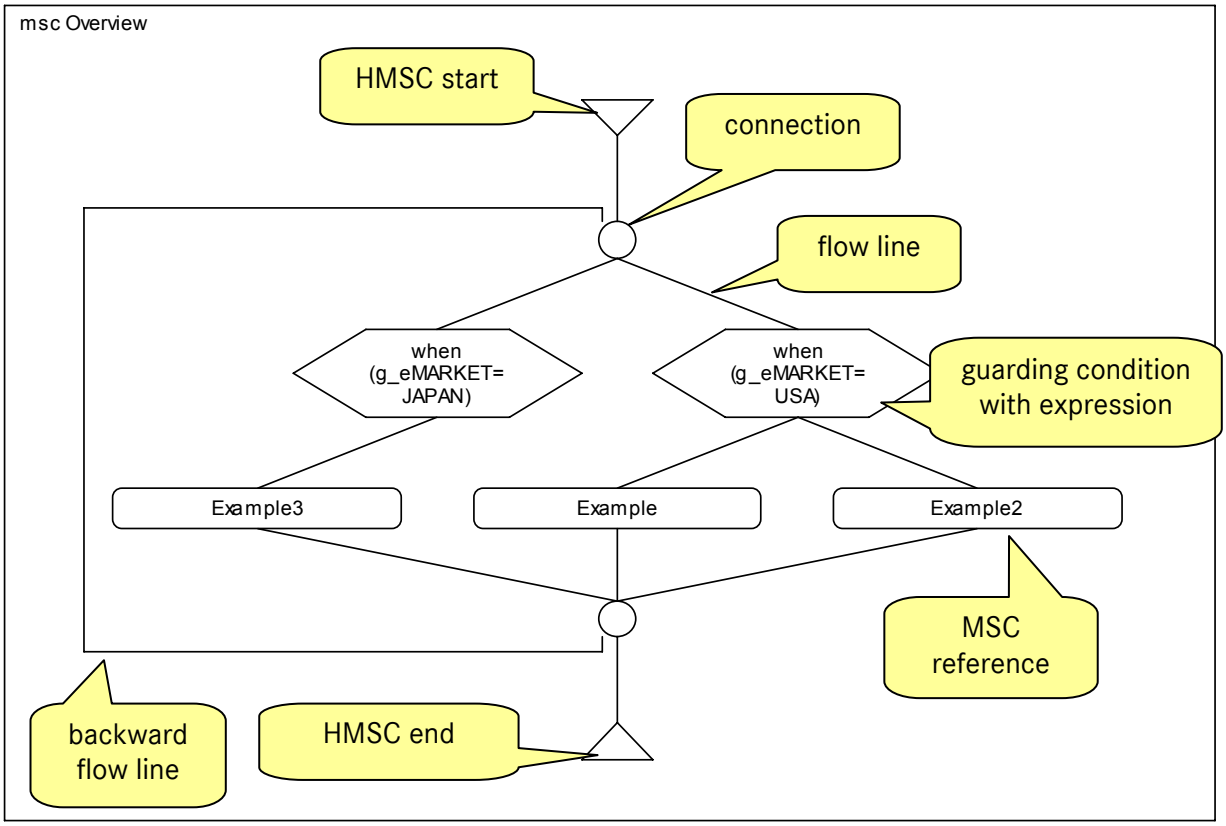
International Telecommunication Union (ITU)

A collection of MSCs is capable of painting an arbitrarily exact picture of the system. MSCs that describe recurrent communication behavior are to be reused by the mechanism of referencing. The MSC standard explicitly supports a modular structure of MSCs. There are two different categories of MSCs: HMSCs (High Level MSCs) and Basic MSCs.

Here is an example of a Basic MSC:



While Basic MSCs describe the communication between different parts of a system on message level, High-level MSCs provide a means to graphically define how a set of MSCs can be combined.



When testing telematics and infotainment systems, it is important to be able to determine or control the preconditions to the test in a way that ensures that they are uniform throughout any number of test runs. For example, a mode change from CD to Radio can only be successfully tested when the current mode is the CD mode.

If a scenario is present in the form of MSCs, and we want to test that particular scenario, we have to find a way to guide the system to the point where the preconditions are established. This is usually done by means of HMSCs.

The MSC standard proposes two flavors of MSCs, the graphical notation (MSC-GR) and the textual (MSC-PR) notation. Both notations are semantically identical.

3 Specific Adaptation of MSCs for Telematics and In-Car Infotainment

The MSC2000 [1] recommendation deliberately does not prescribe the communication protocol implemented in the specified system. For the application in the automotive field, provisions have to be made to embrace the different communication busses and relevant protocols.

MSCs of the telematics/infotainment system describe the bus communication of the bus systems involved. In addition to the bus communication, communication patterns between software components inside a device may be specified in some cases as far as they are relevant for system behavior.

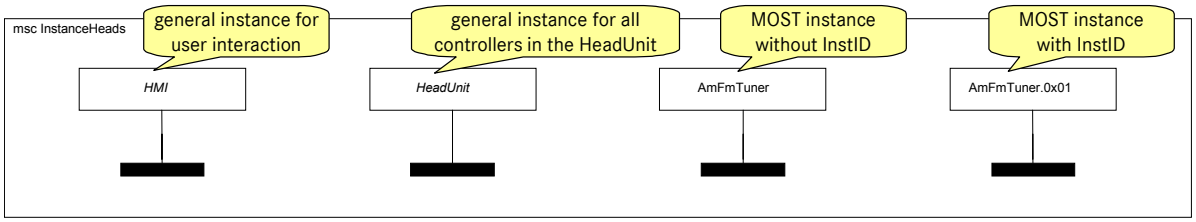
MSCs that are strongly related are grouped to a *scenario* (e.g. a scenario concerning HMI events for the radio application). Scenarios that belong together are grouped to a *scenario category*.
During the adoption of MSCs in the telematics and infotainment domains, it became evident that the current MSC standard is not sufficient to fully meet the needs of those who create specifications in the MOST domain, especially when complex data types are involved. Therefore “named parameters” have been introduced. Please see “Appendix A: Extension to MSC2000 (Named Parameters)” for more information.

3.1 Instances

Instances are represented by an instance head box, a vertical axis and an instance end symbol.
Instances in telematics/infotainment MSCs may equally be MOST FunctionBlocks, MOST FunctionBlock Controllers or even CAN nodes in the system. Examples of legal instance names in the MOST domain are NetBlock, AmFmTuner, and PowerMaster. Those instances that implement a MOST FunctionBlock should always bear its name. Controllers of FunctionBlocks are not required to bear the name of the FunctionBlock; in fact, they shall never bear its name.

MOST instances may be specified in three different ways. The FunctionBlock name (FBlockID) can solely identify the instance or alternatively be accompanied by the Instance-ID (InstID). Furthermore there is a user-defined variant available (General Instance), i.e. an instance that cannot be represented by a FunctionBlock.

Instances in MOST



3.1.1 The HMI Instance

The Human Machine Interface (HMI) instance is a so-called “user-defined instance“. The complement of a user-defined instance is a “MOST instance“. MOST instances are associated with MOST FunctionBlocks, while user-defined instances are not.

Our interpretation of the HMI instance is that we imagine the user, who stimulates the HMI, to be a part of that instance, as if he was sitting inside. We also do not name the actual buttons and switches the user operates, but rather add an abstraction layer that translates the operations of the user into events, which the user can create or receive. So even if the user navigates through a number of menus of his HeadUnit to start a search for the next radio station, we would only model one input from the HMI instance to the system, for example named “TuneNextStation“.

For the HMI messages we have agreed on a notation that lets us easily distinguish those from MOST or CAN messages. The names of messages from the HMI instance are prefixed “MMI_Event“, while message to the HMI instance are prefixed “MMI_Update“. (We assume that the update usually will be perceivable by the user, either by a change of the displayed content or any other form of user notification, or a combination. Still, there is only one message to the HMI instance.)

The second part of the message name contains the name of the application, for example “Radio“. The next part is the abstraction of the user action, which usually maps to a particular function of an application.

The message may also contain parameters. For example, when the user decides to tune to a certain frequency, the frequency would be included as parameter. The frequency could be provided as a numeric value, or as a constant or variable like “MaxFrequency“.

It is important to notice that the parameter interface needs to be consistent for all messages with identical names. This means that the tune-to-frequency message should not occur with a varying number of parameters.
Why is this important? When we deal with MOST messages we can rely on the MOST FunctionCatalog to obtain the exact layout of a message. If we omit a few, we will still know if the provided parameters comply with the FunctionCatalog. The messages which are user-defined, however, in most cases will not be matched against an interface description. So the only reliable information we have comes from the messages themselves, and therefore it is highly recommended to give them a stable interface.

A few examples of HMI messages
Only a trigger event:

- MMI_Event.Radio_TuneSeek

Events with parameters:

- MMI_Event.Radio_TunePreset(i)
- MMI_Update.Radio_Frequency_set(i)

3.1.2 Deployment Information

DeviceID and device names can be used. However, to promote reusability of MSCs for upcoming projects, in general they are left out. Then there is no information in the MSC concerning the device that implements the FunctionBlocks.
Although an abstract view of the system is sufficient during the system architecture phase, it is later required to have a concrete view of the system, i.e. during system integration. So the information about the deployment of the individual logical system parts (FunctionBlocks) to actual devices needs to be injected somewhere between specification and system integration.

3.2 Messages

MOST messages in MSCs should be strictly based on the MOST FunctionCatalog. Even though early design phases do not require a detailed description of parameters, it is mandatory that every MOST and CAN message is matched with the MOST FunctionCatalog or CAN database. Thus, at any stage in the specification process, consistency with the corresponding database is guaranteed.
In addition, the frequent use of the underlying databases will detect errors and increase overall quality, which will be helpful during system integration and generation of test cases.

3.3 Timing

In order to enable an automated test generation, MSCs have to be supplemented with further information. That information is similar to non-functional requirements, as known from requirements engineering. During system operation - and therefore also during test execution – timing is critical. So it is good practice to add timing restrictions and timers as soon as possible in the specification process. This can also prove helpful in finding timing inconsistencies among a number of connected MSCs that otherwise would lead to incorrect implementations.
We propose to add timing constraints whenever standard MOST timings do not suffice.

3.4 ”Processing” Messages

In a MOST environment, ”Processing” or “ProcessingAck” messages can occur after a MOST method has been triggered by ”StartResult” or “StartResultAck” OpTypes. If the device that implements the method cannot answer within a defined timeframe, it will send “Processing” messages to inform the requester that the request is still being processed, but operation has not finished yet.

Processing messages basically could occur in any part of an MSC after the method has been called.

In most cases it is not helpful to include those Processing messages in the MSCs, as they will make the MSC more obscure to human readers. However, it should be kept in mind that they will occur in an actual system, and need to be handled properly.

In some cases it might be required to model Processing messages, i.e. when the receiving MSC instance wants to trigger an animation that informs the user of the ongoing operation, e.g. by displaying an animated hourglass or by offering an opportunity to cancel the operation.

3.5 Notifications

MOST Notifications are used to track changes of properties. When a property is notified, then any change of that property will cause a notification message to be sent to the devices that have request to be notified.

Pertaining to MSC design, Notification messages raise issues, quite similar to the "Processing" peculiarities. Thus, the approach to circumvent potential complexity problems is almost the same.

3.5.1 General Notification Handling

We recommend including the following in one Basic MSC for each scenario or scenario group, depending on your individual requirements:

- Setting of notifications
- The first received Status message for every notified property

This MSC can be reused for different scenarios. Those Notification MSCs should have names that clearly distinguish them from other MSCs, e.g. by prefixing “Notification”. Tools that subsequently transform those MSCs could dynamically determine which notifications are currently active, and set a filter accordingly, to allow notification messages to appear arbitrarily without breaking test cases.

Notifications are most commonly set during system startup and initialization phases. In many cases, you will therefore find Notification MSCs referenced from system or subsystem initialization MSCs.

3.5.2 Notification Messages as Triggers

Apart from the modeling of notification messages as described in "General Notification Handling", notification messages are usually not depicted in the MSCs, **unless** they have impact on the current scenario, e.g. when a thread is waiting for a notification message before it continues operation.

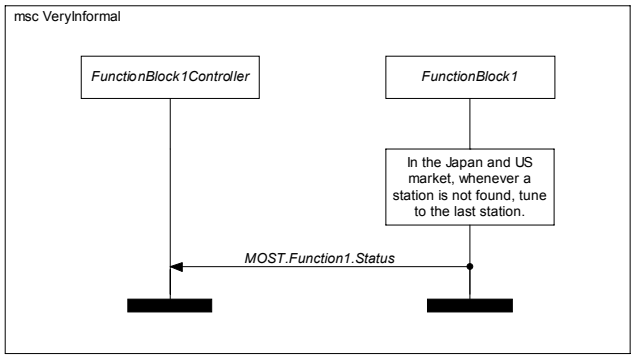
Usually it is not necessary to mark notification messages. If it is required to make the reader aware of the "notification-ness" of the message, a "Notification" comment can be attached to the message.

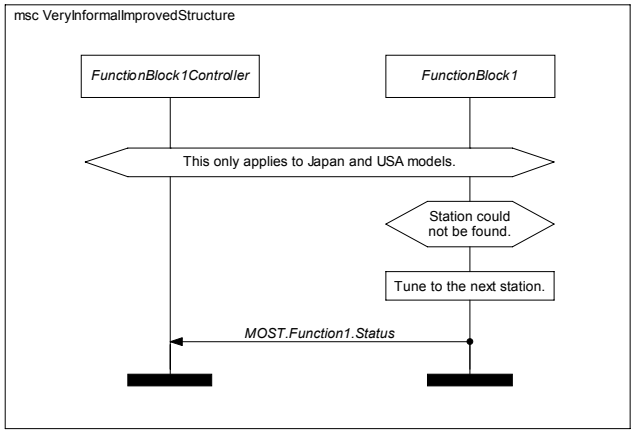
Inclusion of all notification messages at all times is hard to model, although it can be done, e.g. by introducing notification loops in concurrent (see "4.11 Inline Expression: Parallel") MSCs. We have come to the conclusion that this would add information which usually is of no interest to System and Device Architecture, and increase complexity without providing adequate benefit.

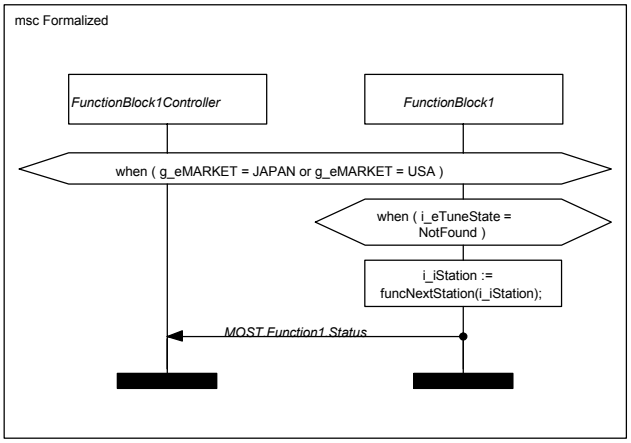
3.6 Formal Conditions and Actions

Conditions and actions will be described in detail in “4 MSC Notation and Guidelines for MSC Composition”. Right now, we will give you a step-by-step example of how to make your MSCs more formal.

We recommend using common language in MSC comments only.

Message Sequence Chart	Comment
	The action is informal. For example, a sentence like “In the Japan and US market, whenever a station is not found, tune to the next station” in an action symbol, is valuable to help the reader understand the intention of the following sequence, but poses many problems to tools that process the MSC in some way. Also, two conditions, namely the restriction to the Japanese and US market, and the fact that a station could not be found are mixed with an action, the tuning to the next station.

Message Sequence Chart	Comment
	The first step to improve the MSC is to differentiate between conditions and actions. Here we have introduced a condition that guards the entire MSC – the applicable markets. Also we have introduced a condition that is local to the FunctionBlock. Here we indicate that the station was not found. The action now contains the tuning part. The problem of the use of common language still persists.

Message Sequence Chart	Comment
	This is the formalized version. We use SDL notation. SDL is very closely related to MSCs, and has also been standardized by the ITU-T as Z.100. We have introduced variables for the market, the current state of tuning and the current station. Also, finding the next station could represent a call to a function that searches for the next available station. The MSC is still comprehensible to the human reader, but very much facilitates subsequent tool-based operations. In cases where the formalization reduces readability too much, comments can be attached to the individual symbols.

The MSC 2000 recommendation strictly separates the MSC domain (structure) and the data domain (data and operations). While the structural elements are clearly defined, no particular syntax, i.e. programming language is enforced on the level of expressions.
In this example, we have used SDL as data language. Depending on different companies' individual needs this could as well be C, C++ or any other programming language. However, we do not recommend mixing a variety of languages. This would take away much of the benefit of formalization.

3.7 Naming Scheme for Constants and Variables

We have found the use of a naming scheme for constants and variables very helpful. It quickly provides the reader of an MSC with type and scope information.

We have gathered our most common uses of type/scope combinations and derived a few simple rules.

3.7.1 Structure

The MSC standard offers no strongly typed data language. We add information about scope and type to constants and variables to help the reader perform visual type checking.

However, you should not rely on the naming scheme to perform automated type checking. Rather, you should utilize the instruments provided by the data language (i.e. keywords for scope and type).
We propose the following format, loosely based on Hungarian notation:

<scope>_<type><Name>
The tokens for scope and type will always be lowercase while the case of the name will depend on whether we are dealing with a constant or variable. Names of constants are always uppercase.

3.7.2 Scope

Our main scope factors are MSCs and instances. The lifetime of constants/variables may be bound to single instances or individual MSCs. You should always make sure that the scope of a constant or variable is explicitly defined!

Please note that including MSCs through references or “using” clauses extends the visibility of variables and constants; contained declarations and definitions will be visible on the level of the referencing MSC and even above! This can easily lead to conflicts. Therefore, we strongly advise you to choose variable names deliberately, especially when you design MSCs that will be reused in many different scenarios.

Token	Scope	Comment
g	Global	Known to all instances in all MSCs. In our SDL-like data language, global variable scope is introduced by the “global” keyword.
i	Instance-local	Known only to one instance but in all MSCs. In our data language, the keyword “process”, followed by an instance name, defines the scope of instance local variables.
l	MSC-local	Known to all instances but only within the scope of one MSC. Whenever formal data parameters are used, those are MSC-local.

3.7.3 Variability

Determines whether the value of a data item can change during its lifetime.

Variability	Comment
Constant	The data item’s value is immutable. Names of constants are uppercase (e.g. LIGHTSPEED).
Variable	The data item’s value is allowed to change. Names of variables can be mixed case (e.g. AverageTravelSpeed).

3.7.4 Types

This is a list of the types that are most common in the telematics/infotainment domain. It is by no means complete.

Token	Type Name	Comment
b	Boolean	Contains either True or False.
e	Enum	Enumeration type; consists of a number of constants.
i	Integer	Contains an integer which is typically in the range -32,768 to 32,767.
r	Real	Contains a floating-point number. We do not make any assumptions on precision or range.
s	String	Contains a variable-length string. Strings contain a sequence of alphanumeric characters.
c	Complex Type	This type can be used whenever non-simple types are referred to. Those can be records, arrays, arrays of records.

3.7.5 Examples

Here we provide a number of example names to illustrate the naming scheme.

Example	Comment
g_bClampState30	We assume that every connected device is aware of the clamp state through some kind of low-level communication which is not part of the MSC specification. This makes the clamp state global (known to all instances in all MSCs). Also, the clamp state can change. Therefore it is variable and written in mixed case. The type we used here is Boolean. It might make sense to use an Enum type instead. The possible values would then be ON and OFF. Consequently, we would name the variable g_eClampState30.
g_eMARKET	We consider the market a globally known constant, so the name is capitalized. The individual values could for example be ECE, USA and JAPAN.
l_iSOME_PARAM	An auxiliary integer constant that is local to an MSC but visible to all instances in that MSC. (The const-ness of such an item might be imposed by the MSC standard when a referenced MSC contains formal parameters.)
i_iDiskNumber	The variable for the slot that contains a CD changer's currently selected disk. If the CD changer is an instance, the disk number is known only to this instance during its entire lifetime in all MSCs.
i_iCAPACITY	The nominal capacity of a CD changer is also local to its instance, but unlike the disk number it will not change.

4 MSC Notation and Guidelines for MSC Composition

4.1 Instance

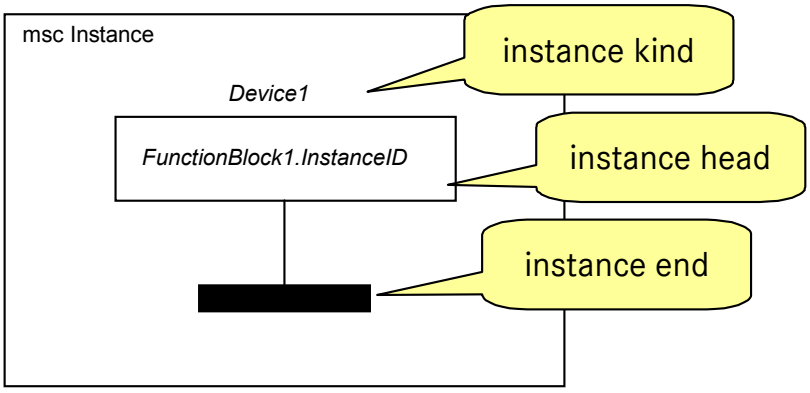
Instances represent the communication partners in the different buses involved in the telematics/infotainment system. Instances are represented by vertical axes in the MSC. An axis starts and ends with a non-filled and a filled rectangle respectively.

MSC instances represent time as follows:

- time flows from top to bottom
- there is no global time scale
- there is no common time scale for different instances

The MSC standard states:

A global clock is assumed for one Message Sequence Chart. Along each instance axis the time is running from top to bottom, however, a proper time scale is not assumed. If no coregion or inline expression is introduced a total time ordering of events is assumed along each instance axis.



In general, we do not use dynamic instance creation. Therefore all instances start to live on the top of the MSC and end at the bottom of the MSC.

The example above shows an MSC instance which has the instance name “FunctionBlock1.InstanceID“. The instance name is always depicted inside the instance head. In the MOST world it consists of FunctionBlock name and instance ID that are separated by a period.

An instance may also have an instance kind. In combination with SDL, instance kinds can be systems, blocks, processes or services. In the MOST world it is not uncommon to deviate from the SDL usage and provide device information in the instance kind field. In the example we have used this to indicate that FunctionBlock1.InstanceID will be deployed to Device1.

Individual enterprise regulations determine whether instance kinds are used during the specification of telematics/infotainment devices. In the MSC Cookbook we normally will not use instance kinds.

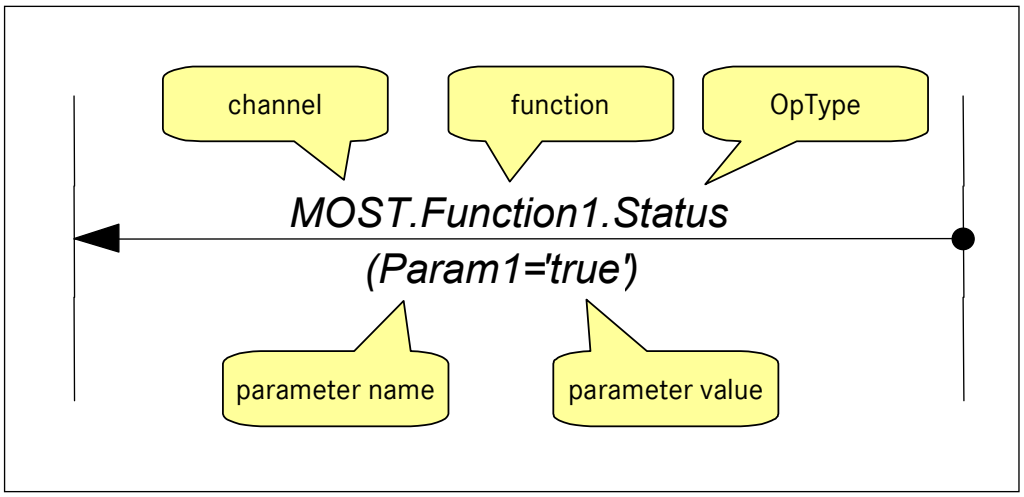
Apart from MOST instances, general instances, as described in chapter “4.1 Instance” can be used, too.

4.2 Message

4.2.1 Overview

Arrows, directed from the sending to the receiving instance, denote communication.

The label on an arrow denotes the message exchanged by the two instances involved. The message name is to be placed above the arrow, whereas the parameter(s) are optional and placed below the arrow in parentheses.



The given example shows a MOST message which requires some additional explanation:

The message name consists of three parts: The channel name, the function name and the OpType, separated by periods. Unlike the channel name, function and OpType require no explanation as they are well known in the application domain. Typical channels names are “MOST”, “USERDEF” and “CAN”. Those are required to facilitate synchronization of the specification with databases or catalogs that list all available messages. So a tool that parses an MSC will always know in which database to look for a certain message to either support the user in creating an MSC or verifying an existing MSC against a database of messages.

Messages that belong to the “USERDEF” channel usually will not be matched against a database.
In the MOST domain we use message parameters in a special way, which is further explained and justified in “Appendix A: Extension to MSC2000 (Named Parameters)”. Right now it should be enough to know that “parameter names“ or “named parameters“ are an extension to the MSC 2000 recommendation. The use of named parameters gives us the freedom to specify only those parameters which are relevant to a certain use case, while the MSC standard requires the use of positional parameters which would force us to supply all parameters at all times for reasons of consistency.

Something that is not visible in the graphical representation, but becomes evident when we look at the textual representation is that a message consists of two parts, namely sending and receiving:

```
FunctionBlock1: label L0; out MOST.Function1.Status,1(Param1='true') to
FunctionBlock1Controller;

FunctionBlock1Controller: in MOST.Function1.Status,1(Param1='true') from
FunctionBlock1;
```

This makes it possible to model message overtaking. However, as we usually work with a synchronous transmission technology, we highly recommend not using message overtaking.

Some readers of this document might wonder what "`Status,1`" stands for in the MSC snippet given above. The "`,1`" is a so-called "message instance name", represented by a number in this case.
Message instance names are used to distinguish multiple occurrences of an identical message. This is something that the MSC standard requires.
Most existing MSC tools will always add such a name, even if only one instance of that message exists.
Also, there is a "`label L0`". This label is important for providing an anchor for timing restrictions.

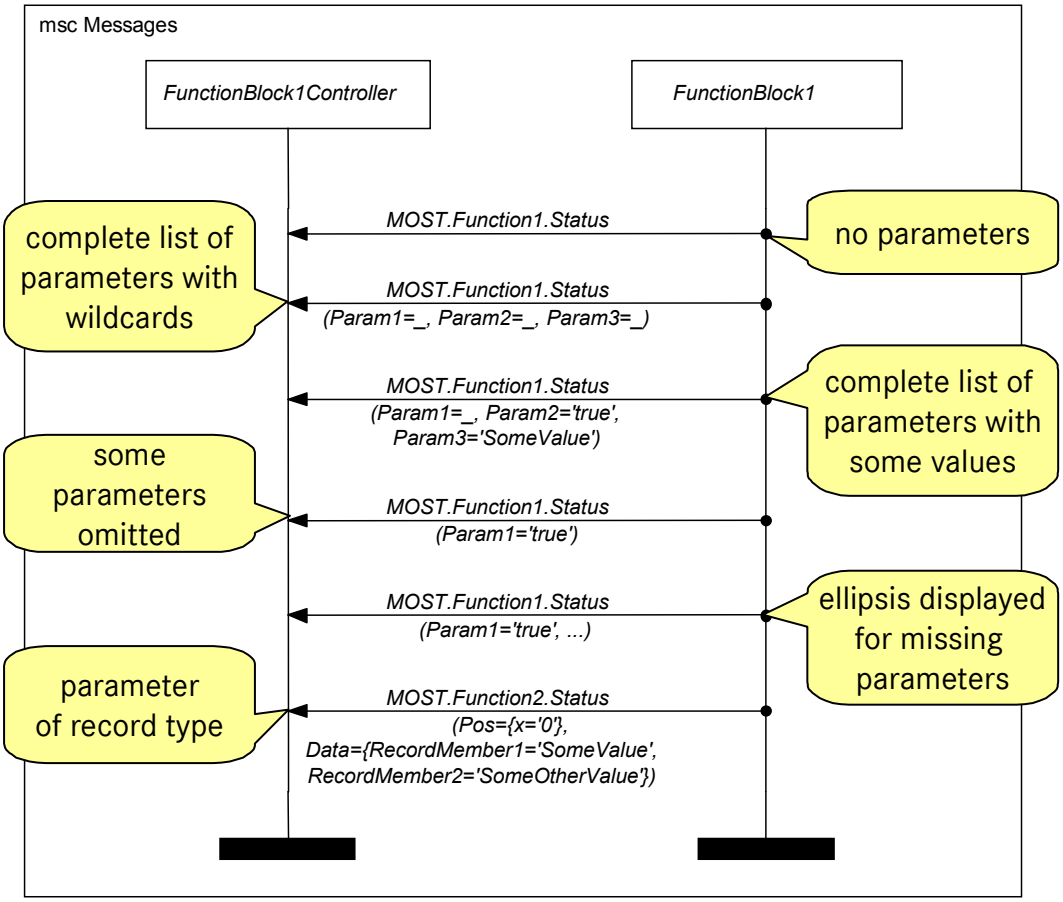
4.2.2 Message Parameters

The current parameter concept for the MOST domain does not fully comply with the MSC standard. Please see the "Appendix A: Extension to MSC2000 (Named Parameters)" chapter for a detailed explanation of the extensions.
If parameter values are not known, an underscore (_) will be used as wildcard character. A wildcard character limiting a stream, array or record represents possibly following parameters.

Parameter Examples

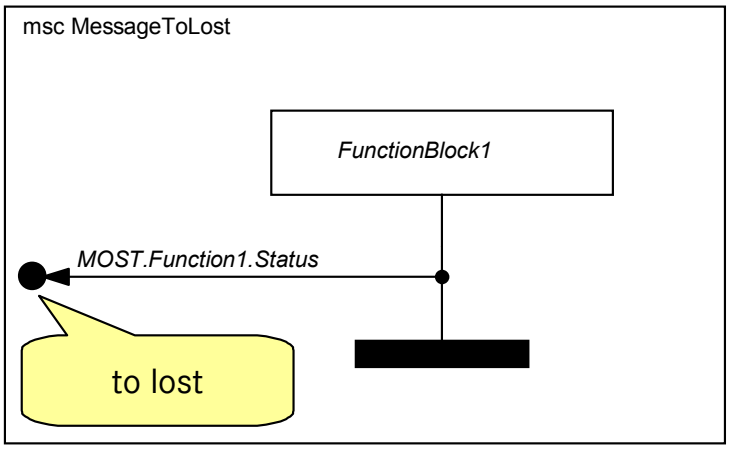
Parameter	Description
Number = _	Any value is legal for the parameter "Number".
Number = 3	Only "3" is a legal value for the parameter "Number".
Number =3=: A	This example shows the use of named parameters and variables. The parameter with the name "Number" has a value of "3". This value is assigned to the variable "A". Variable "A" can later be reused, e.g. in a conditional expression.

Variants of parameter specification and definition



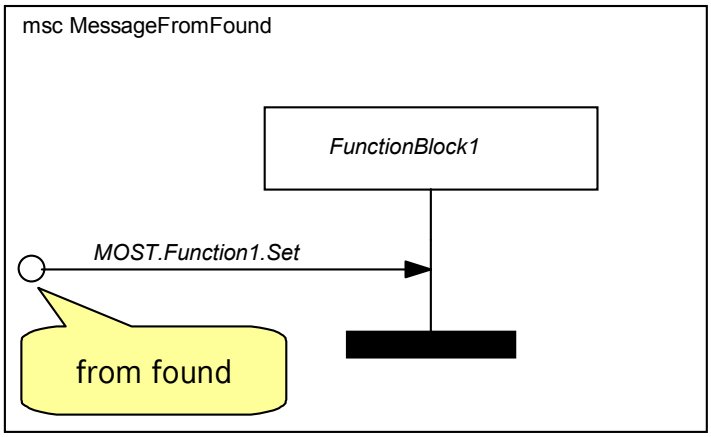
Messages can be specified in different ways according to the level of detailed required. The level of detail can be rendered more and more precisely over time during the development process.

4.2.3 Lost Message



Currently, the “message to lost” construct is one of the possible ways to describe the sending broadcast messages. Please see “4.14 Broadcast/Groupcast” for more details.
Whenever possible, do not use messages to lost. See the next chapter for an explanation.

4.2.4 Found Message



Currently, the “message from found” construct is one of the possible ways to describe the receiving of broadcast messages. Please see “4.14 Broadcast/Groupcast” for more details.

Messages to lost or messages from found have one particular feature which is not beneficial for our application: Either the sender or the receiver is unknown. Thus, such a message only consists of one line in the textual representation:

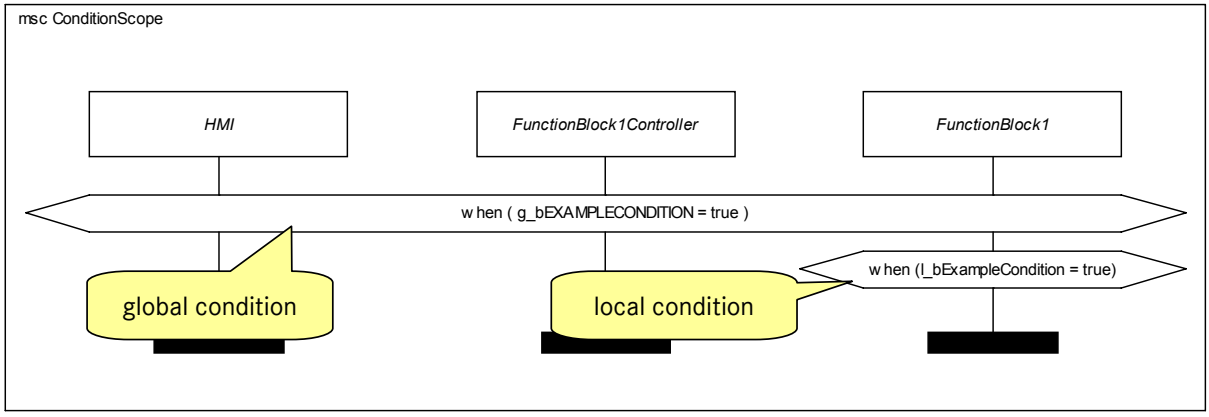
```
FunctionBlock1: label L0; in MOST.Function1.Set,1 from found;
```

This makes it very hard to resolve them when using automated tools to process the MSCs.

4.3 Condition

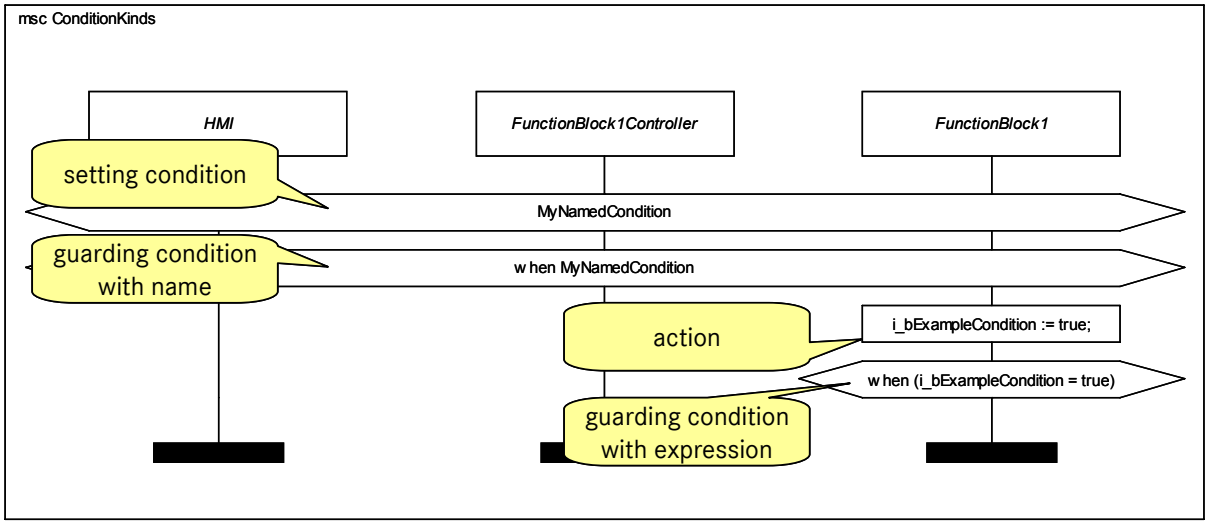
4.3.1 Condition Scope

Conditions may apply to one instance (local condition), to some instances (non global condition) or to all instances (global condition) of the MSC. It is important to choose the proper scope for a number of reasons, which we will explain after an introduction to condition types.



4.3.2 Condition Types

There are two types of conditions: *setting* and *guarding* conditions.



It is important that we differentiate between named conditions and conditions with expressions.
Take a look at the above MSC: The first condition is a **setting condition**, which means that MyNamedCondition is set, so that it will evaluate to “true” whenever checked. The second condition is a **guarding condition** which checks the state of MyNamedCondition. The following parts in the MSC will be executed if it evaluates to “true“, which it does in this case.
Both are “**named conditions**“, and the names are within the domain of the MSC standard.

Named conditions have a few particularly displeasing characteristics:

- When a condition is set, all other conditions are deleted.
- Named conditions cannot explicitly be unset.
- Complex data types cannot be used.
- Logic operations are not allowed

The last condition is a **guarding condition with an expression**. Expressions are outside the domain of the MSC standard. That means that different groups of MSC users can apply their own programming or data languages. In this example we have chosen an SDL-like form. To cause the expression to evaluate to “true” we use an action symbol, where we explicitly set the ExampleCondition variable to true.

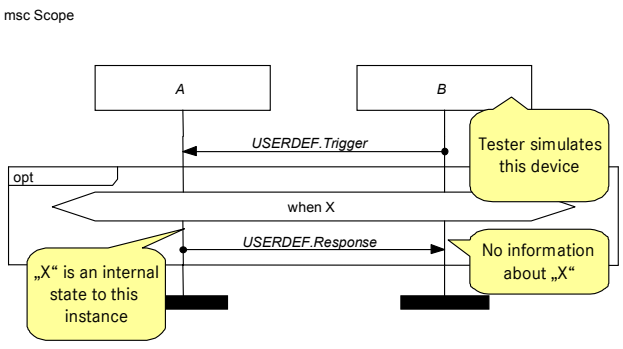
We recommend the use of conditions with expressions for the following reasons:

- Unlike named conditions, they allow the use of complex data types.
- Variables can be set to arbitrary values without influencing other variables.
- Logic operations can be used.

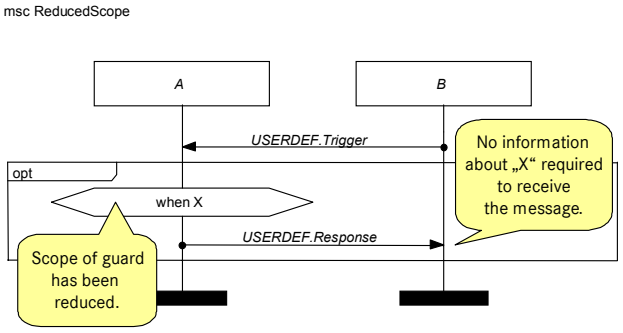
4.3.3 Overscoping of Conditions

The general rule is: Reduce the scope of conditions as much as possible.
However, there are good reasons to use globally scoped conditions. For example, certain features of a device that are local to a market have no relevance in other markets. Therefore it makes sense to guard an entire MSC with a constant that represents the market, and ensure that only features pertaining to a particular market are considered.

To understand why other conditions should be local, you have to think like a test engineer. The test engineer usually cannot monitor the inner state of devices. If the sending of a message from device A to device B depends on a certain state of device A, let us call the state X, the test engineer can deduce that state by observing the message, but he **cannot anticipate it**.

Message Sequence Chart	Comment
	Now let us assume we perform a black box test of device A, so the test equipment simulates device B. If both sending and receiving of message “Response” is guarded by X, it means that the test equipment is not allowed to consume that message if the state variable evaluates to X! But the test equipment does not have any knowledge about the current value of the state variable. So what happens now?

A human will be able to resolve that problem and realize that the condition was not supposed to apply to device B. But as soon as we introduce tools that transform specifications into tests we have a real problem.
Apart from that it is also confusing for the readers of the MSCs in the specification phase, as in some cases it will not be clear which instance “owns” a condition, leading to misinterpretation and faulty system behavior.

Message Sequence Chart	Comment
	The modified MSC has clearer semantics. Now, the sender will send the message in state “X”, and the receiver will be ready to receive it, not having to know about the internal state.

Apart from keeping the scope of conditions minimal, you should only apply conditions when they have an influence on the behavior of the system. Conditions that only serve the purpose of describing an assumed state of the environment without having direct influence on the behavior as described in the MSC should be transformed into text boxes.

4.3.4 Conditions in HMSCs

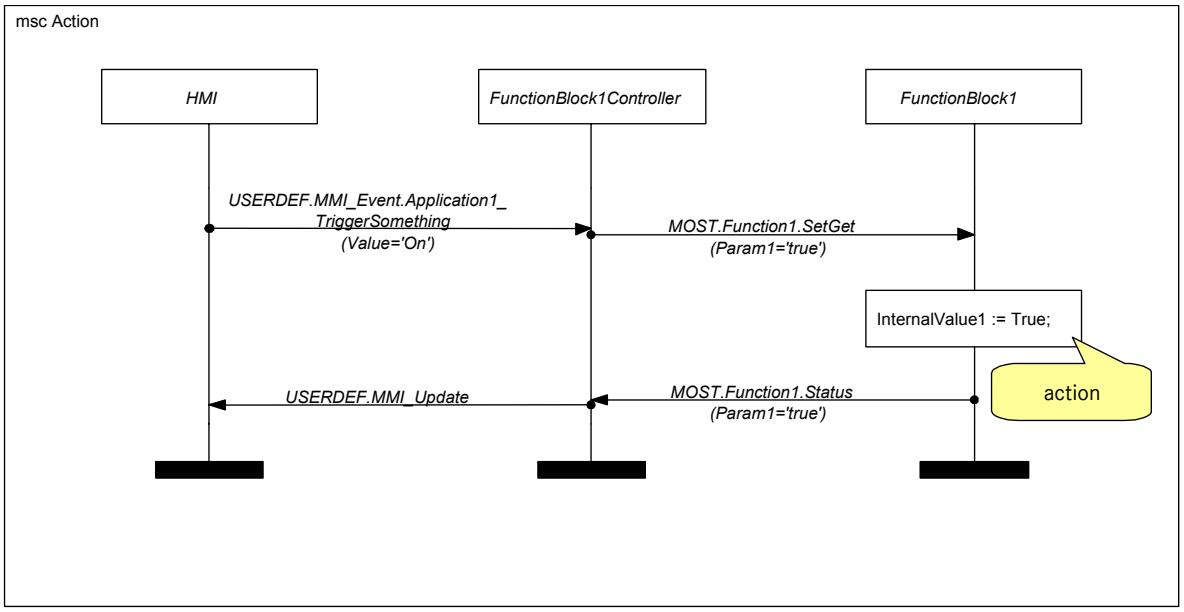
The MSC standard is quite rigorous and does not allow data-based conditions in HMSCs at all. Mainly, this is due to the lack of instances in HMSCs. Thus, it is impossible to determine the context of instance-local data.
However, the nature of telematics and infotainment system specifications makes us highly dependent on data. Therefore, we have decided to allow conditions in HMSCs but restrict them to global variables.

Conditions in HMSCs must not rely on data that is local to instances!

4.4 Actions

An Action describes the internal activity of an instance.

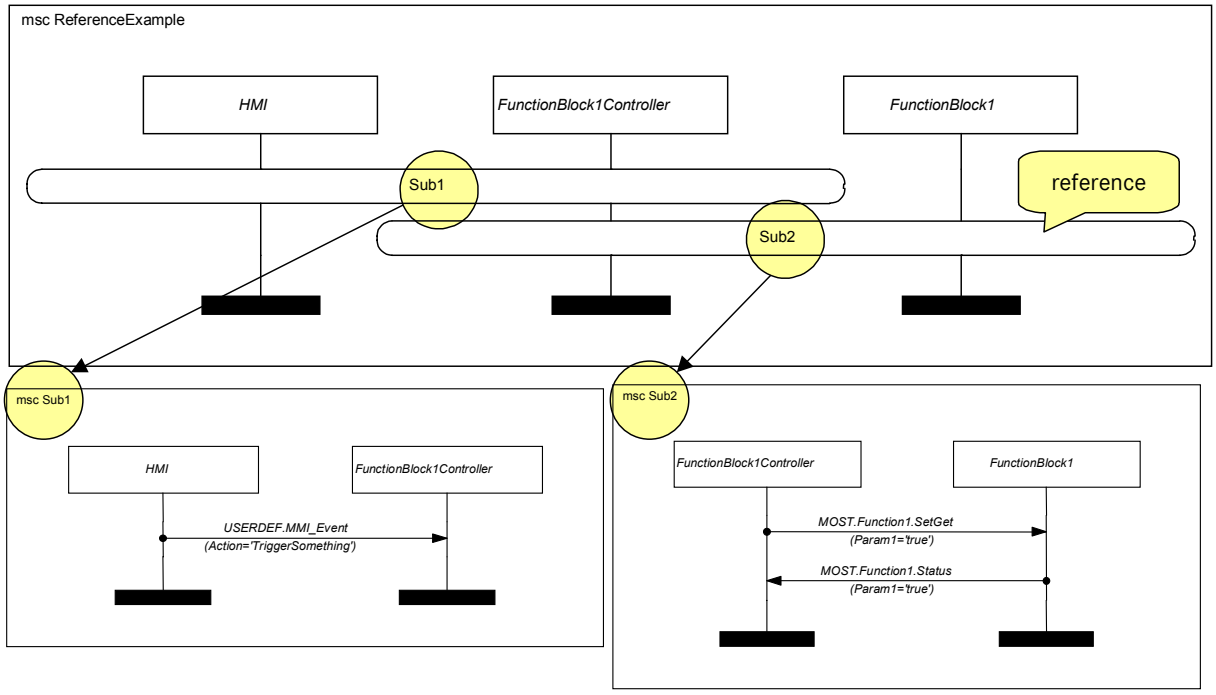
The MSC standard does not make any assumptions on the data language, leaving it up to the user whether informal text or a formal language is used to describe the semantics of an action. We propose to decide on a programming language that will be used consistently throughout an entire project, for both actions and conditions. We have used SDL notation in this example.



The MSC standard determines that multiple data statements in an action are evaluated concurrently. To prevent side effects, we suggest that you do not use more than one statement per action!

4.5 Reference

The reference within an MSC defers to a different MSC which has to be uniquely identifiable in the collection of MSCs.



4.5.1 Requirements

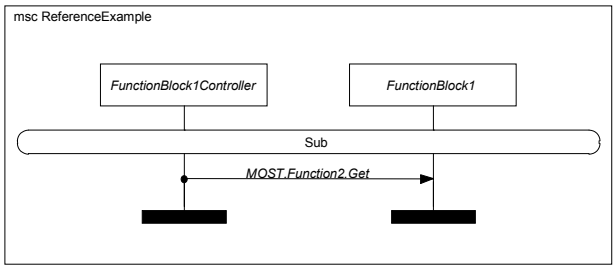
MSCs must not refer to themselves, be it directly or indirectly.

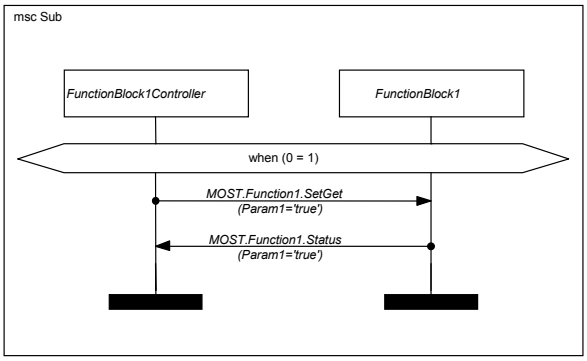
The reference symbol must include all instance axes that occur in the referenced MSC. It may overlap instance axes that do not occur in the referenced MSC.

If the MSC contains at least two references: All instance axes shared by at least two of the referenced MSCs must be present in the referencing MSC. This ensures the assignment of an order to the events on axes that occur in referenced MSCs.

4.5.2 References to Guarded MSCs

The MSC standard does not explicitly define the semantics for references to MSCs which contain global guards. Let us take a look at the following situation where “ReferenceExample” references “Sub”.





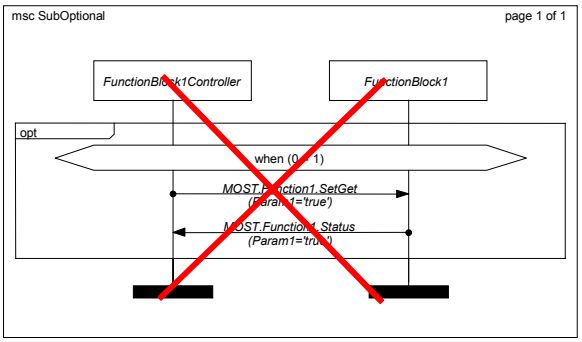
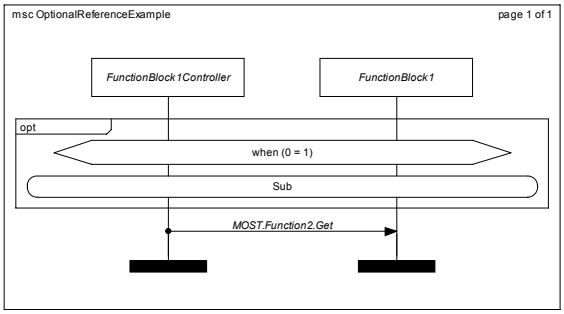
The guard in MSC “Sub” is always false, so none of the messages in Sub will be sent. In MSC ReferenceExample, will FunctionBlock1.Controller send Function2.Get or not?

The MSC standard itself is inconclusive. Two interpretations are possible:

- The referenced MSC is considered optional and Function2.Get is sent.
- The referenced MSC blocks the referencing MSC. The entire scenario becomes dynamically illegal.

At first glance, the idea of considering the referenced MSC optional appears favorable because the scenario will execute no matter what.
However, assuming implicit options leads to another question: How do we model the case when the scenario **must** become illegal? The condition may be a critical one and the system might not be able to operate when it is false.

The blocking interpretation is to be preferred. Whenever a referenced MSC is meant to be optional, the reference shall be explicitly modeled that way!
We do not recommend making all referenced MSCs themselves optional by introducing a bounding optional box. Rather we recommend controlling their uses as shown below.
Always try to place the optional box in the outermost possible context! For reasons of clarity, we also propose to repeat the guarding condition in the referencing MSC:

Not recommended!	**Recommended!**
	

4.6 Environment and Gates

Gates represent the interface between the MSC and its environment.

The MSC standard mentions several types of gates, namely

- Message gates
- Order gates
- Create gates
- Gates on inline expressions

We are mainly interested in **message gates**. The other gate types are not widely used, and we have not encountered a case where they would have been required, so far.

4.6.1 Environment

The term “environment” covers everything outside an MSC, which is able to send messages into the MSC, or receive messages from the MSC. The concept of an MSC environment can refer to two different situations:

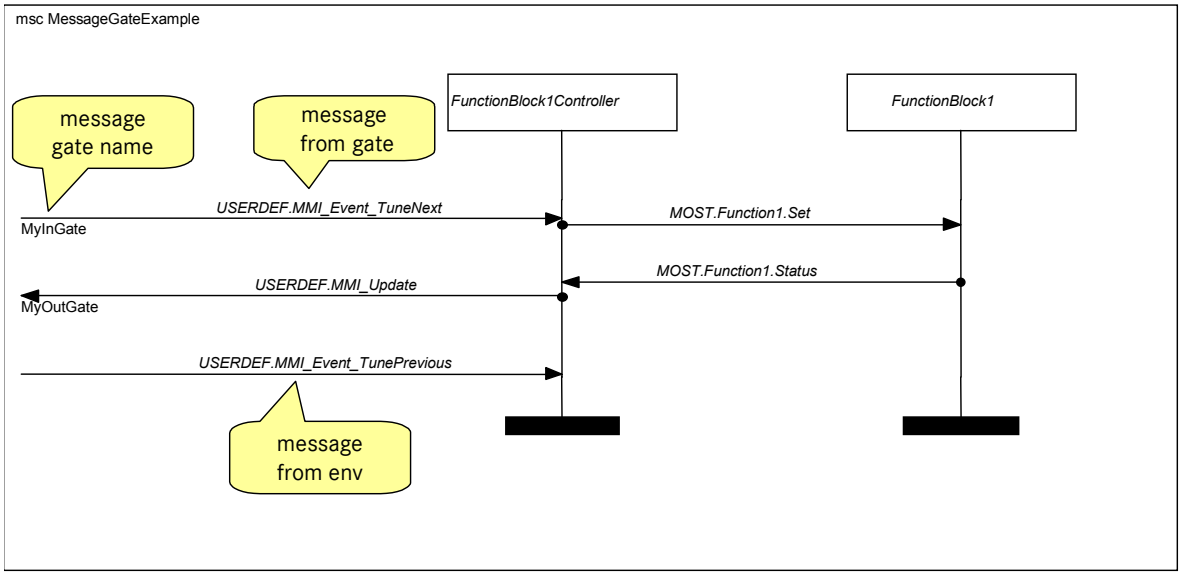
- The environment of a referenced MSC (i.e. interaction with some other instances that are still within the boundaries of the system being modeled by MSCs)
- The environment of the whole system(.i.e. any interactions between the system and the outside world)

Unless stated otherwise, the environment we speak of is the environment of the MSC.
An MSC communicates with the environment through gates. Gates always have a name, although it is allowed to create MSCs where the gate name is not explicitly given. In that case the gate name is implicitly composed of the message direction (in or out) and the message name.

Incomplete messages, i.e. those messages that are sent but never consumed (lost messages), and those messages that are found spontaneously (found messages), are not part of the communication with the environment. In lost or found messages either the receiver or the sender is not known, while the use of the environment demands that both sender and receiver of a message are specified. It is important to understand that instances and gates are both equally qualified to be sender and receiver of messages.
Therefore, a message from a gate or a message to a gate is considered a complete message.

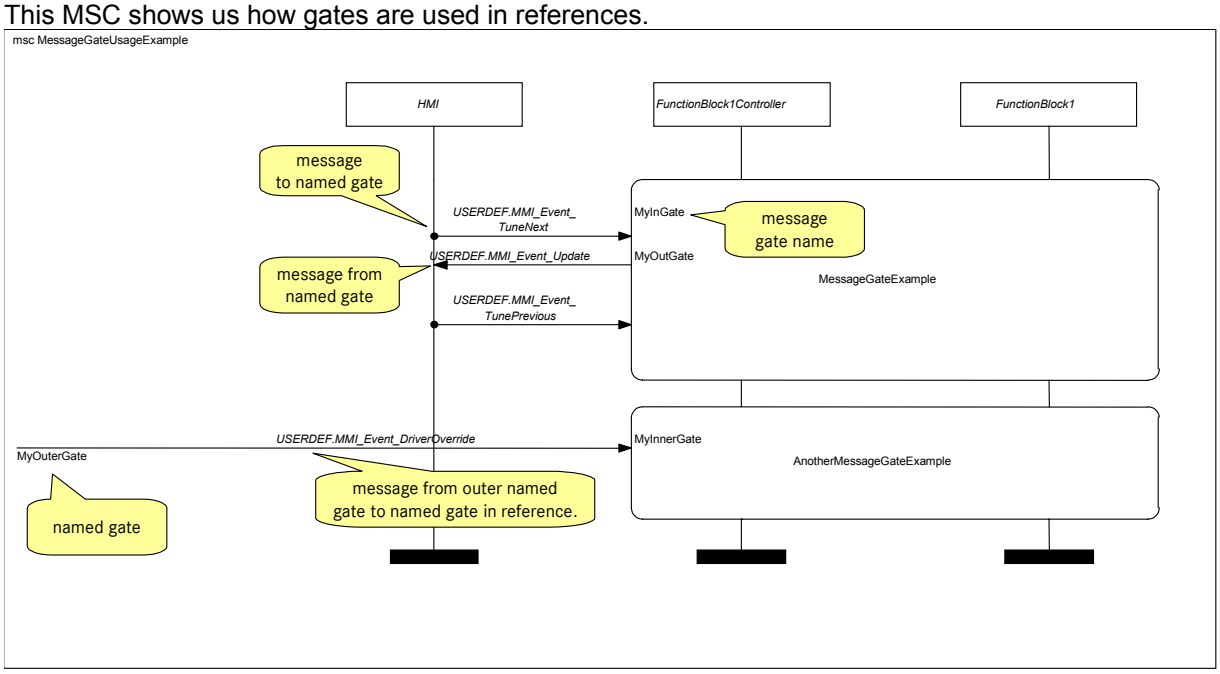
4.6.2 Message Gates

This MSC illustrates the use of explicitly and implicitly named gates. Messages from implicitly named gates are called “messages from env”.



4.6.3 Use of Message Gates

This MSC shows us how gates are used in references.



Here, the “MessageGateExample” MSC from the previous chapter is referenced. It is easy to see how the referencing MSC and the referenced MSC come to match when you take a look at both.

You can also see an additional reference to an MSC named “AnotherMessageGateExample” (the content of which we did not depict here). The interesting part about it is that messages which come from the environment can be forwarded to a reference. No MSC instance is involved in the communication, but merely two gates – one that sends the messages, and one that receives it.

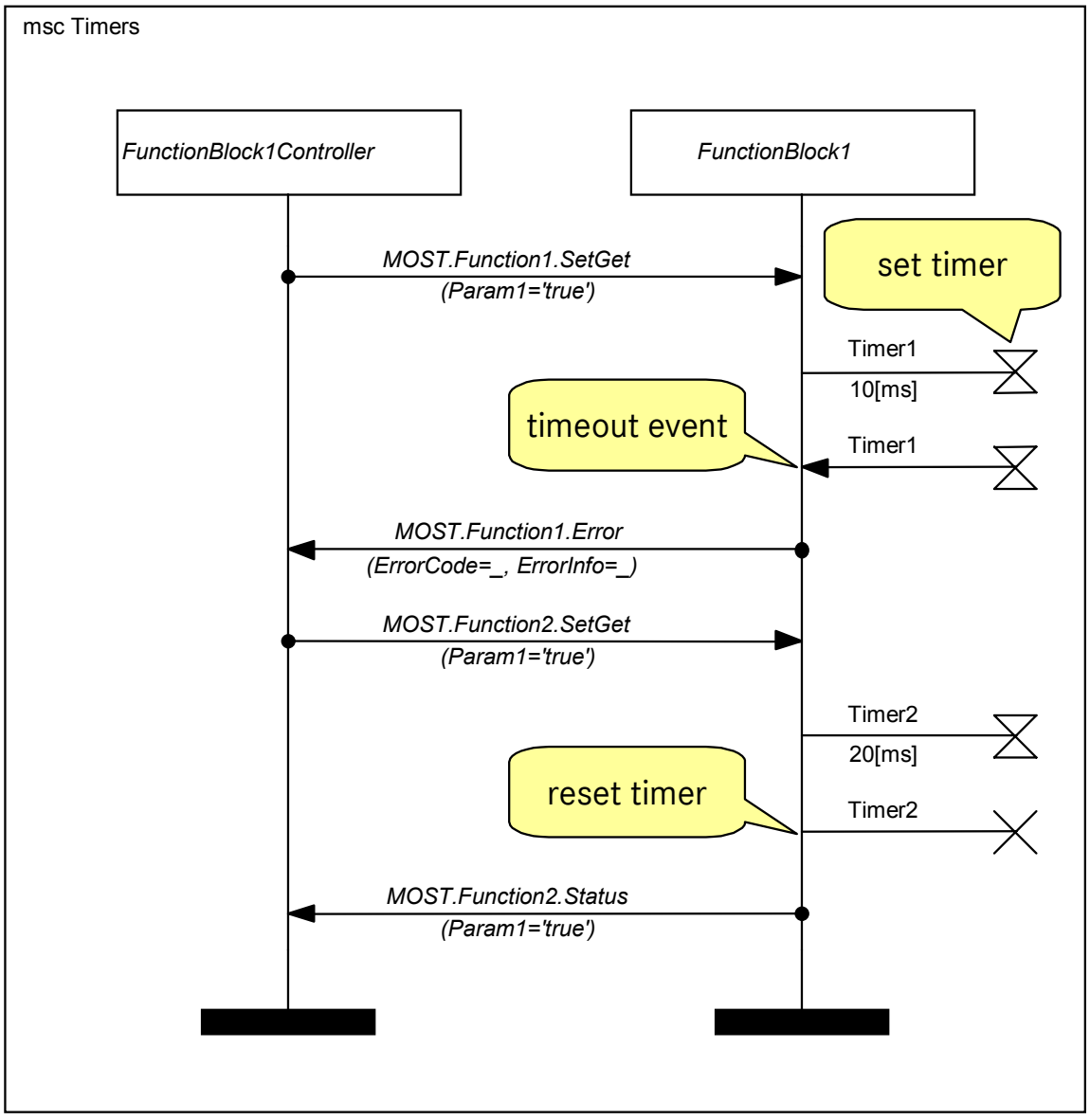
So, why do we need gates at all? We have two main applications for gates. One is the reuse of generic MSCs, which combines well with gates. Also, gates are essential in our favored approach to modeling Broadcast and Groupcast messages.

When using gates, it is important to ensure proper matching of the in- and out-gates.

You will find more information on generic MSCs and handling of Broadcast and Groupcast messages in the following chapters.

4.7 Timers and Time Constraints

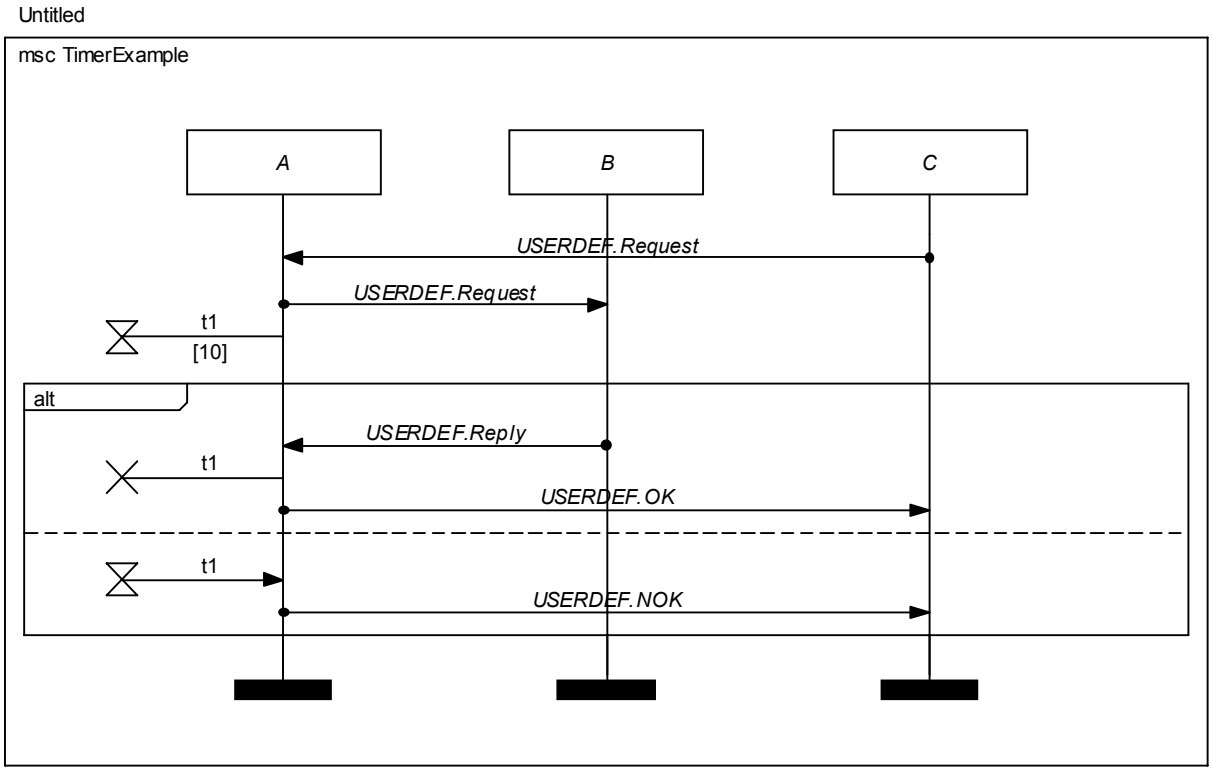
Timers provide information on the timing requirements which have to be respected to achieve correct (free of errors) execution of the MSC. Timers can be set and stopped. When a timer times out, a “timeout event” occurs.



The above MSC example uses the “MSC 96” form of timers.

Timers can be used for the following two purposes:

- specify delays
- specify an alternative to receiving a certain message within a certain tie interval (the timeout becomes a “no interaction” kind of event)



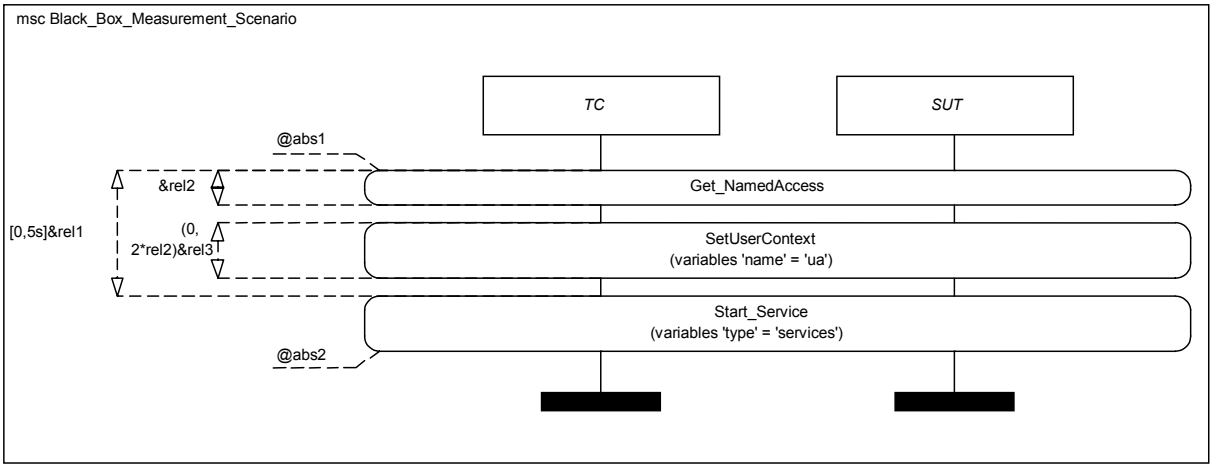
This example shows how timers can be used to specify behavior that depends on not receiving a certain message.

Instance A that is forwarding a request for instance C, waits for the “Reply” message from instance B.

If it arrives, everything is fine, timer t1 is stopped and instance C is notified of success. If nothing happens, in the second part of the alternative, timer t1 will time out and a “not ok” message is sent to tell instance C that B did not reply in time.

When the duration between two events is of interest, we propose the use of the more intuitive way of describing time constraints, which has been introduced in MSC 2000.

Time constraints do not affect the execution of MSC events. Still, for traces to be valid, the timings as specified have to be respected.
On the other hand, timers are real events!

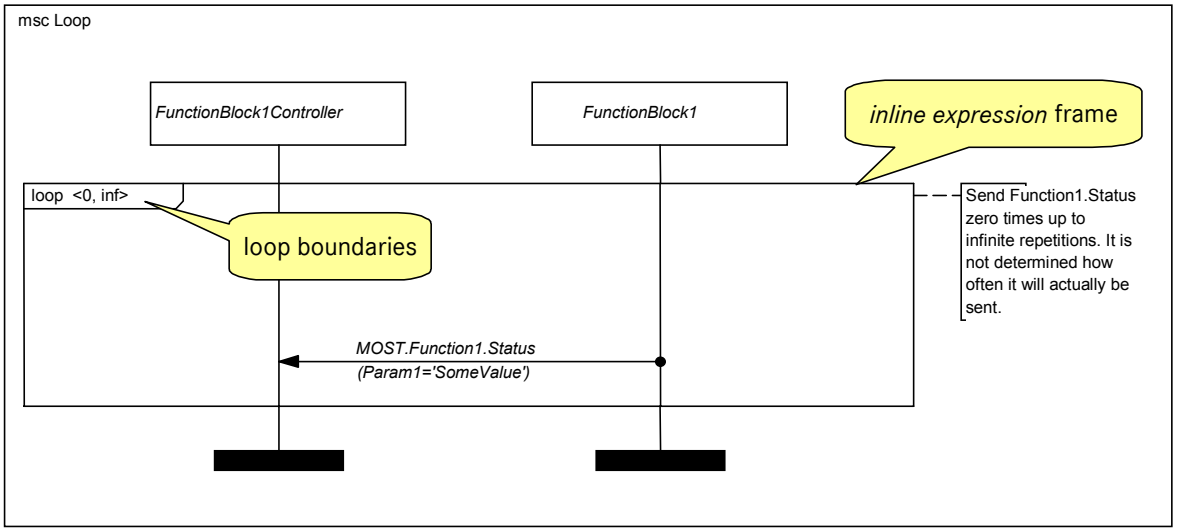


MSC 2000 timing example from the MSC-Specification:
The MSC Black_Box_Measurement_Scenario *uses absolute measurements (indicated by @) and relative measurements (indicated by &). Absolute measurements observe the value of the global clock.*
The absolute measurements @abs1 *and* @abs2 *give the absolute start and end of the test described as a sequence of* Get_NamedAccess*,* Set_UserContext *and* Start_Service*. In addition, relative measurements are used to measure the time distance between pairs of events.*
For example, &rel1 *measures the distance between the start event of* Get_NamedAccess *and the start event of* Start_Service*. This measurement is combined with a constraint* (0,5s] *meaning that the duration between start of* Get_NamedAccess *and start of* Start_Service *is constrained and the time it takes really (within the given bounds of the constraint) is observed by means of the relative measurement.*
(0, 2*rel2) &rel3 *is also a relative time constraint combined with a relative measurement. This constraint refers to a measurement on the duration between start of* Get_NamedAccess *and its end taken before.*

4.8 Inline Expression: loop

4.8.1 Loop Boundaries

When “loop” inline expressions are used, it is vital that the proper loop boundaries are specified.



The most basic form is “loop <n,m>” where n and m are expressions of type natural *number*s. This means that the operand may be executed at least n times and at most m times. The expressions may be replaced by the keyword inf, like “loop <n,inf>”. This means that the loop will be executed at least n times. If the loop bounds are omitted like in “loop”, it will interpreted as “loop <1,inf>”.
The termination of loops may depend on parameter values. We will provide examples of how to specify those cases.
The loop inline expression includes messages that occur in a loop according to the condition given in the upper left corner of the inline expression.

Loop examples

Expression	Description
loop <a>	Interpreted as loop <a,a>, exactly a repetitions.
loop <a,b>	At least a repetitions / max. b repetitions.
loop <a,inf>	At least a repetitions / max. unlimited repetitions.
loop <a,a>	Exactly a repetitions.
loop <0,inf>	No specified number of repetitions.
loop <inf>	**Caution!** “loop <inf>” means the same as loop <inf,inf>, which is a **truly infinite** loop! This loop cannot be broken!

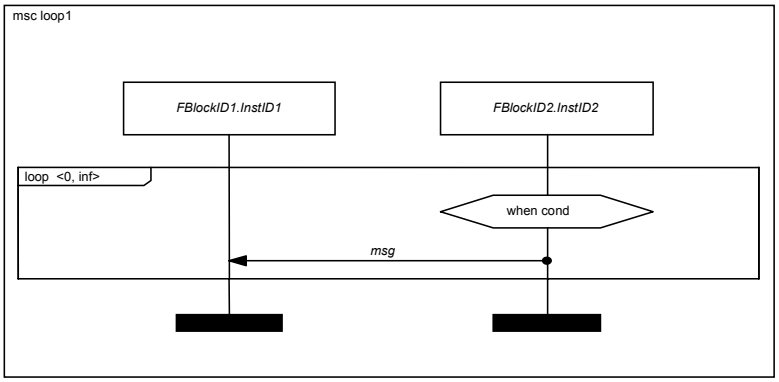
4.8.2 Guarded Loops

When the loop operand is guarded, the loop is terminated if the guard is false, or continued if the guard is true as long as the upper bound has not been reached. Thus, a loop equals an empty MSC if the guard is false the first time the loop is entered.

If the lower boundary of the loop is not reached due to the guard, the entire loop is interpreted as dynamically illegal.

If the number of loop cycles is not bounded, the guard condition can be used to terminate the loop:

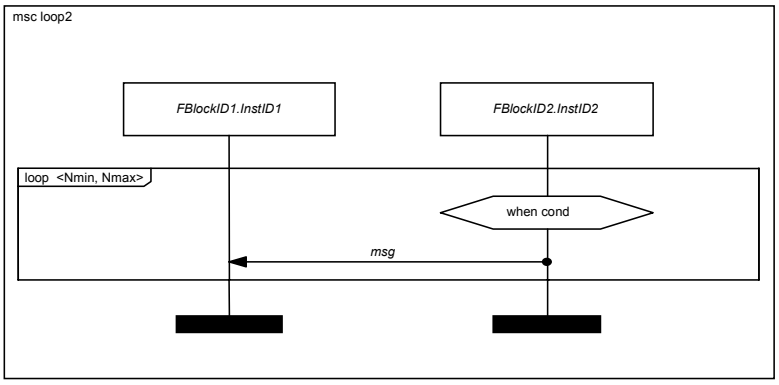
Message Sequence Chart	Symbolic Notation

	`while (cond)`
	`{`
	`  msg();`
	`}`

However, you should be aware that the guard is not altered anywhere in this MSC! So to terminate the loop, we have to assume that while the loop is being executed, a parallel MSC performs the invalidation of “cond”, so that the loop can terminate. What we have here is a potentially infinite loop–which could be exactly what the creator of the MSC wanted to specify.

The following loop shall be executed at least Nmin times, and no more than Nmax times. If the maximum number of loop cycles equals Nmax, the termination of the loop depends both on Nmax and the guard condition:

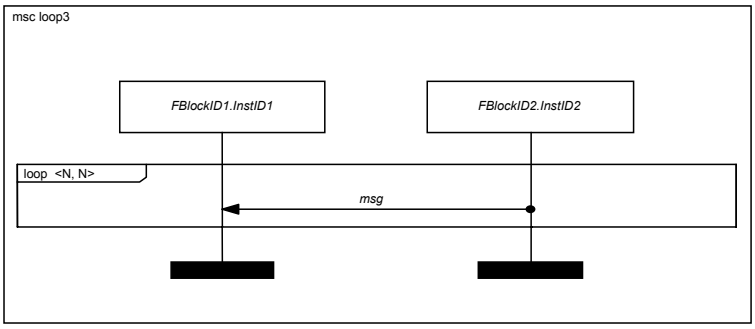
Message Sequence Chart	Symbolic Notation

	`int n = 0;`
	`while (cond && (n<Nmax))`
	`{`
	`  msg();`
	`  n++;`
	`}`
	`if (n < Nmin)`
	`  throw DynamicError;`

In the example given above, there is potential problem. A situation might occur, where the loop counter is less than Nmin, but “cond” does not evaluate to true anymore. So even if the minimum required number of loops iterations has not been reached, the loop is terminated. In the MSC world this is called a “dynamic error”, or as mentioned before the loop might be “dynamically illegal”.

Please also note that “cond” is not altered in this MSC, just as in the example before. In general, it is good practice to do that in the same MSC and not in another MSC, which performs the re-setting in parallel.

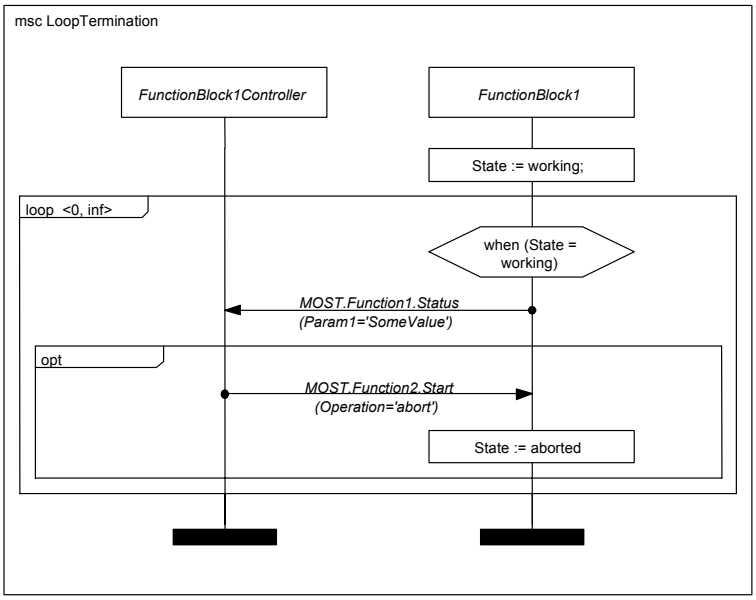
If the minimum and maximum number of cycle loops is identical, it does not make any sense to guard the loop operand:

Message Sequence Chart	Symbolic Notation
	


```
int n = 0;
while (n < N)
{
      msg();
      n++;
}
```

4.8.3 Loop Termination

Here is an example of how to break a loop:

Message Sequence Chart	Symbolic Notation
	


```
StateType State =
working;

while (State == working)
{
   Status();

   if   (ReceivedStart
                (abort))
   {
        State = aborted;
   }
}
```

Please note that we are using “guarding conditions with expressions” in this example, this gives us the opportunity to define actual types. This would not be permitted if we used named conditions.
We have a variable “State” that monitors the current state. Before the loop is entered it is set to working, so the loop will perform at least one iteration. After sending the Status message, it is checked whether or not a “Start” message has been received from the Controller, containing the “abort” as requested operation.

If the message exists, the State variable will be set to “aborted”. Thus, when the loop entry condition is checked the next time, the loop will break as intended, and the Status message will no longer be sent.

4.8.4 Waiting Condition

There are cases when an “MSC shall wait” for a system event to occur that is not explicitly modeled. Such an event might be the change of a clamp state.

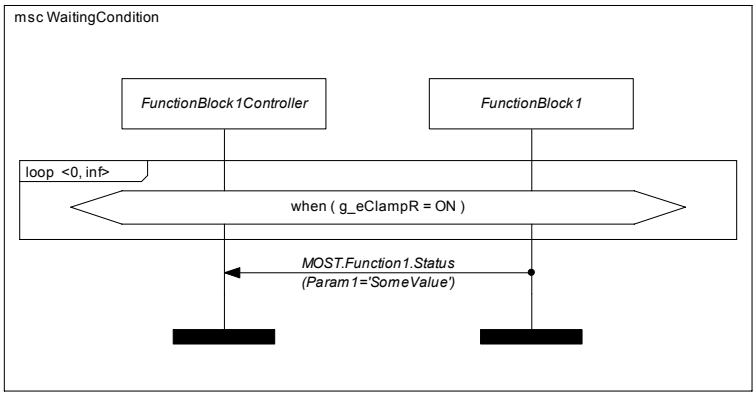
For example, we might want to model the following requirement:

FunctionBlock1 shall **wait** until clamp “R” has value “OFF”. Then it shall send “Function1.Status” to its controller instance.

If we merely used a guarding condition that checks if clamp “R” is OFF, it might evaluate to false when the MSC becomes active, causing the scenario to fail even if the clamp state changes to OFF shortly after. We could explicitly model the change of the clamp’s state as an MSC message. But usually we will not want to ornate application level MSCs with low level communication.

Our solution to this problem can be found in the following diagram. Here, the MSC will wait in the loop until g_eClampR no longer contains ON as value.

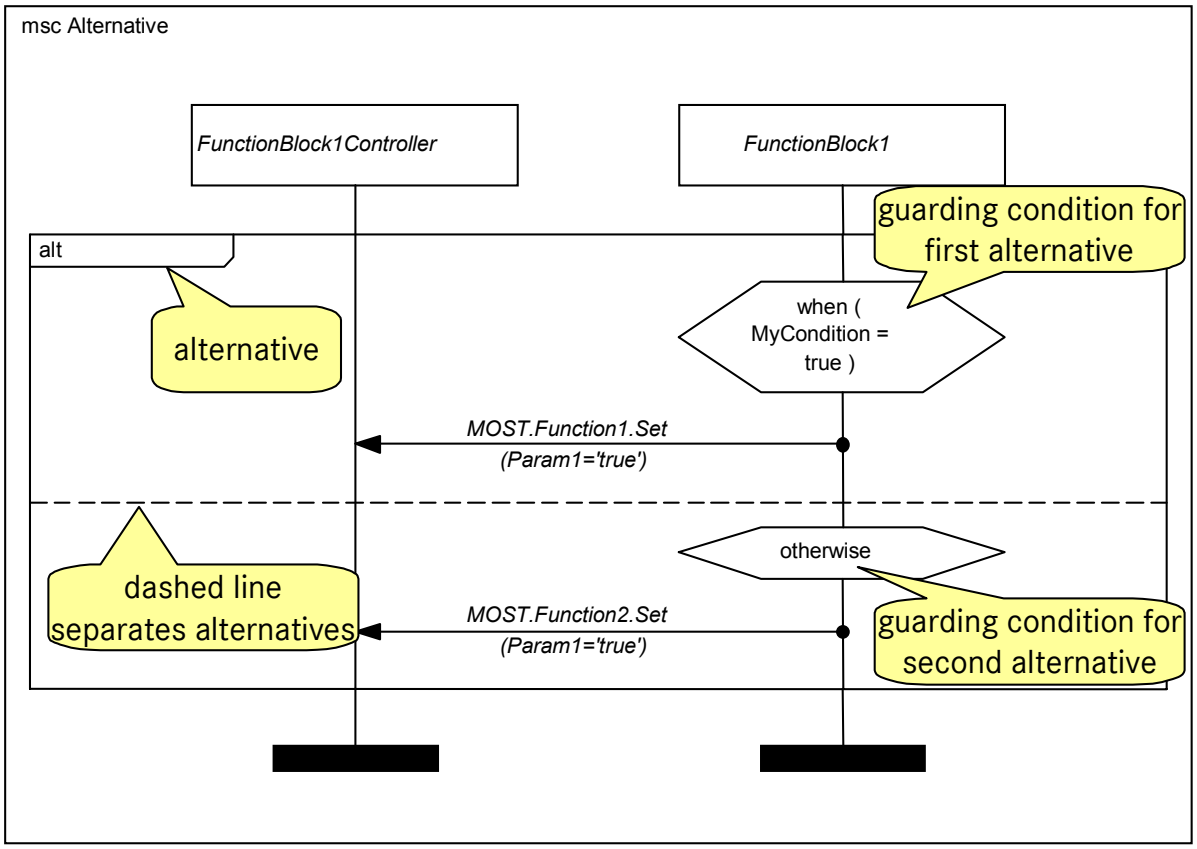
Message Sequence Chart	**Symbolic Notation**

	`while (g_eClampR == ON)`
	`{`
	`  // wait`
	`}`
	`Status();`

Unfortunately, it is not self-explanatory that g_eClampR can alter its state without any visible interaction. We highly recommend to document in detail that variable g_eClampR is expected to be changed **externally**. Otherwise the readers of the MSC will look for the change of clamp state in other MSCs and not find it.

4.9 Inline Expression: Alternative

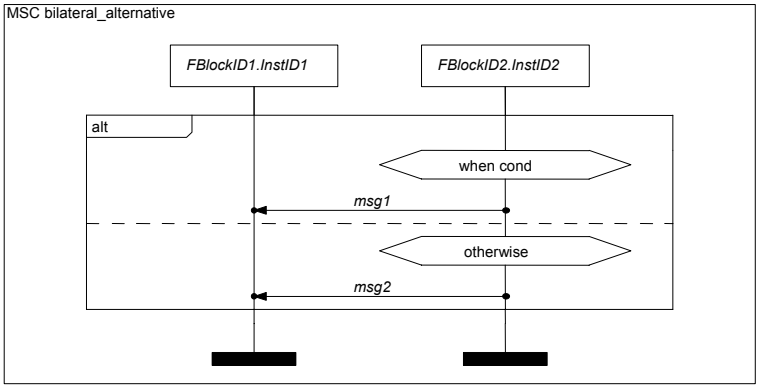
The Alternative inline expression embraces messages which are variants of valid message patterns. Alternative Inline Expressions can be recognized by the word *alt* in the upper left corner of the inline expression. In addition a condition can be placed inside each alternative of the alternative inline expression. A dashed line separates the different alternative areas.



4.9.1 Deterministic Alternatives – Bilateral Alternatives

This means that the message msg1 is executed if the condition cond is true. Otherwise the message msg2 is executed.

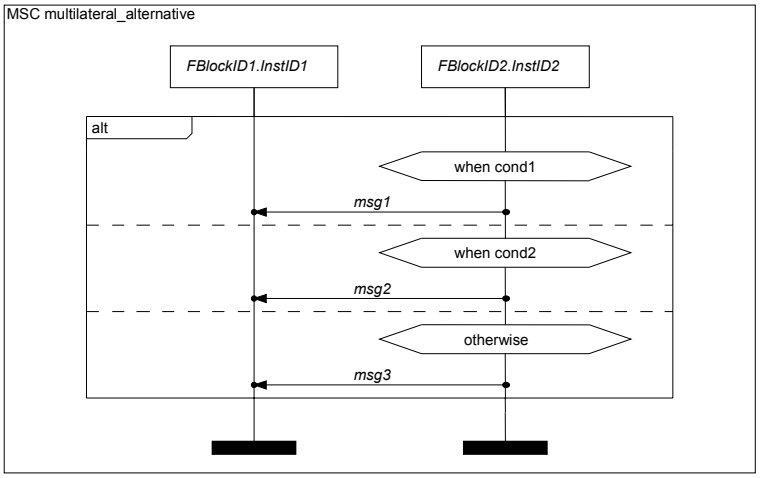
Message Sequence Chart	Symbolic Notation

	`if(cond)`
	`  msg1();`
	
	`else`
	`  msg2();`

4.9.2 Deterministic Alternatives – Multilateral Alternatives

This means that the message msg1 is executed if the expression cond1 is true. If the expression cond1 is false and the expression cond2 is true the message msg2 is executed. Otherwise the message msg3 is executed.

Message Sequence Chart	Symbolic Notation

	`if(cond1)`
	`  msg1();`
	`else if (cond2)`
	`  msg2();`
	`else`
	`  msg3();`

If one alternative block contains one or more unguarded alternatives, those are considered to be "true" by definition, and the "otherwise" branch will never come to execution!

(See ITU-T Z.120, chapter 7.2 "Inline Expression", page 74 "Semantics")

Recommendations for alternatives:

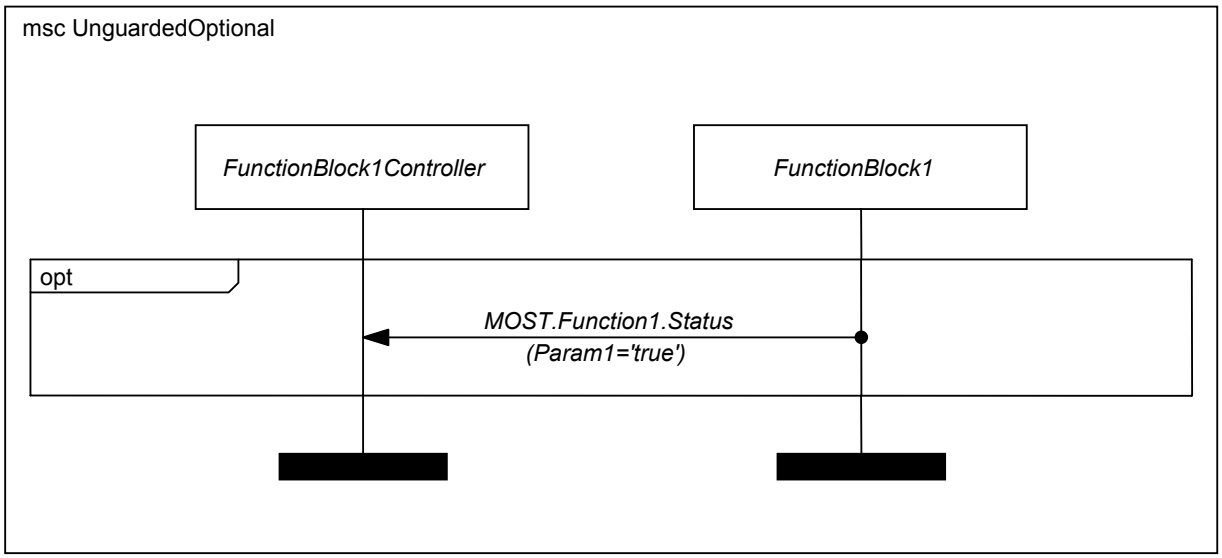
- **Do not mix unguarded and guarded branches. Guard all branches or none.**
- **To avoid non-determinism, do not use overlapping conditions. Keep conditions disjoint among one another.**
- **Do not overscope guarding conditions.**
- **Place guards on the sender instance, not the receiver instance.**

4.10 Inline Expression: Optional

The Optional Inline Expression embraces messages that can occur (without guards) or are necessary under certain conditions (with guards).

4.10.1 General Use

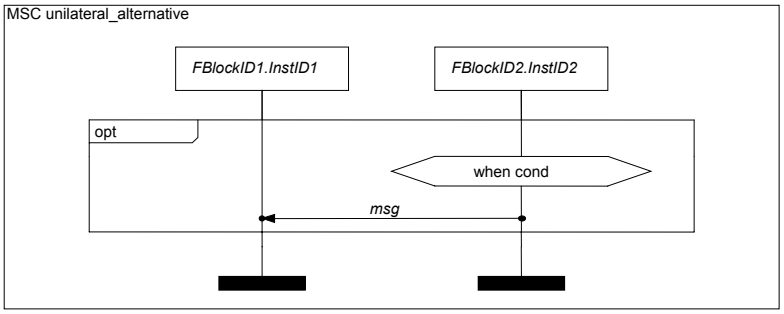
Optional inline expressions are marked by the word *opt* in the upper left corner of the inline expression. In addition, a condition can be placed inside each alternative of the optional inline expression. If the condition evaluates to “true“, the content of the optional expression **must** be executed.



4.10.2 Deterministic Alternatives – Unilateral Alternatives

In the unilateral alternative the execution of one or more messages depends on a condition cond. The condition cond is included in a condition symbol and can be attached to one or more instances.

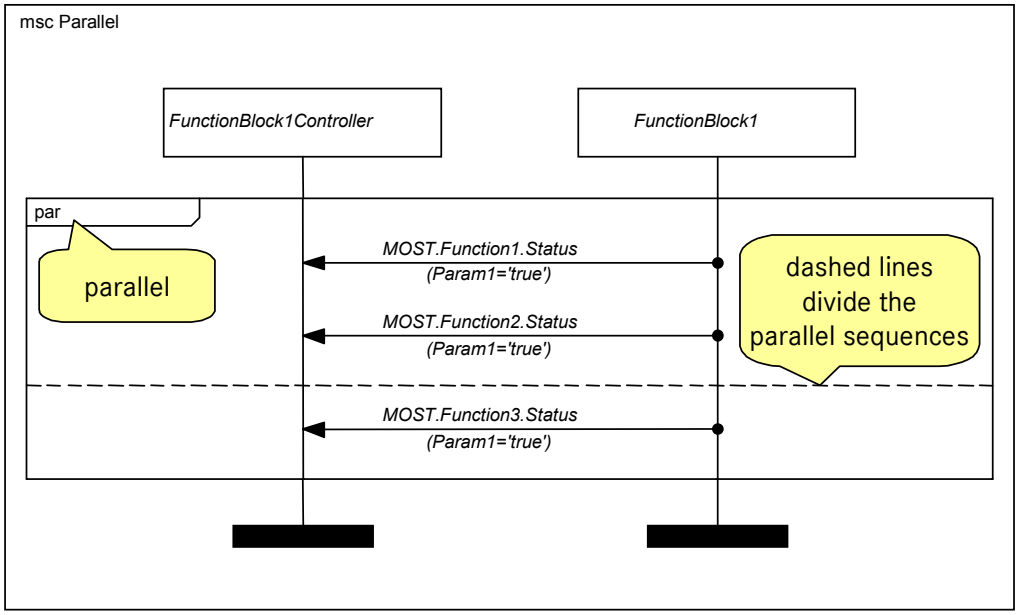
Message Sequence Chart	Symbolic Notation

	`if(cond)`
	`{`
	`  msg();`
	`}`

This means that the Message msg only occurs if the expression cond is true.

4.11 Inline Expression: Parallel

The Parallel Inline Expression includes messages that occur in parallel. Parallel Inline Expressions are indicated by the word *par* in the upper left corner of the inline expression area. A dashed line divides the areas that are to be processed in parallel.

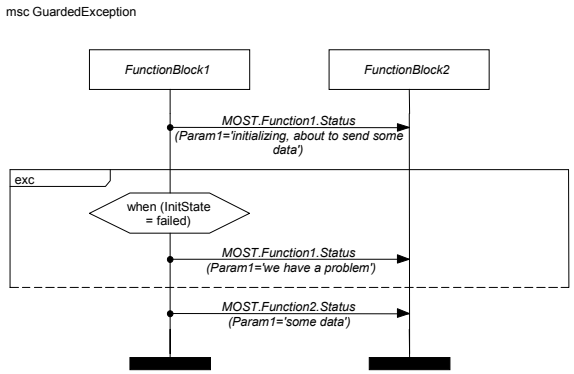
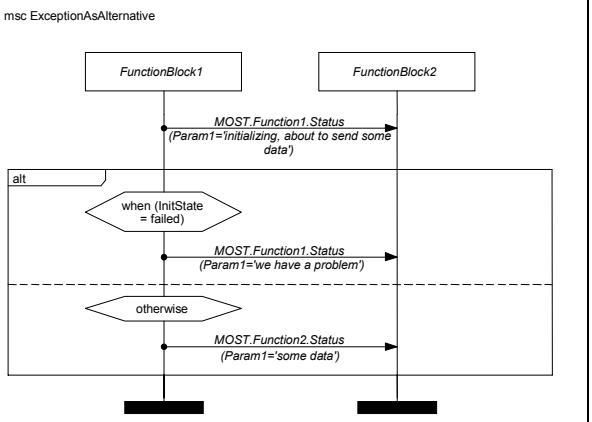


4.12 Inline Expression: Exception

The exc operator is a compact way to describe exceptional cases in an MSC. The meaning of the operator is that either the events inside the <exc inline expression symbol> are executed and then the MSC is finished or the events following the <exc inline expression symbol> are executed. The exc operator can thus be viewed as an alternative where the second operand is the *entire* rest of the MSC.

If an MSC that contains an exception is referenced by another MSC, the messages that follow the reference are **not** influenced by the presence of the exception.
All exception expressions must be shared by all instances in the MSC.

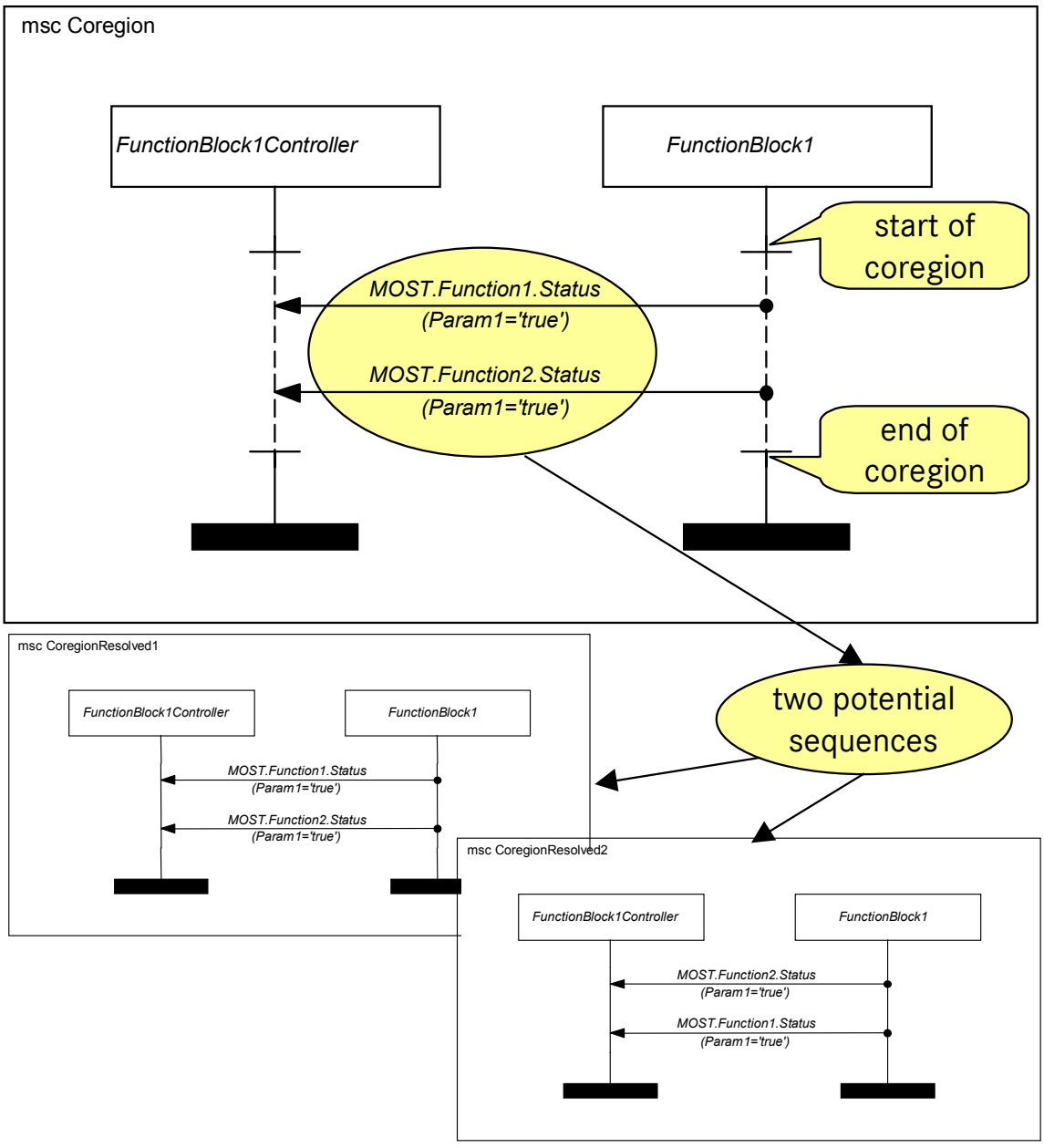
Guarding can be used to give an entry condition for the exception expression. The two following MSCs show a guarded exception expression and the equivalent bilateral alternative.

Message Sequence Chart with exception	Equivalent representation with alternative
	

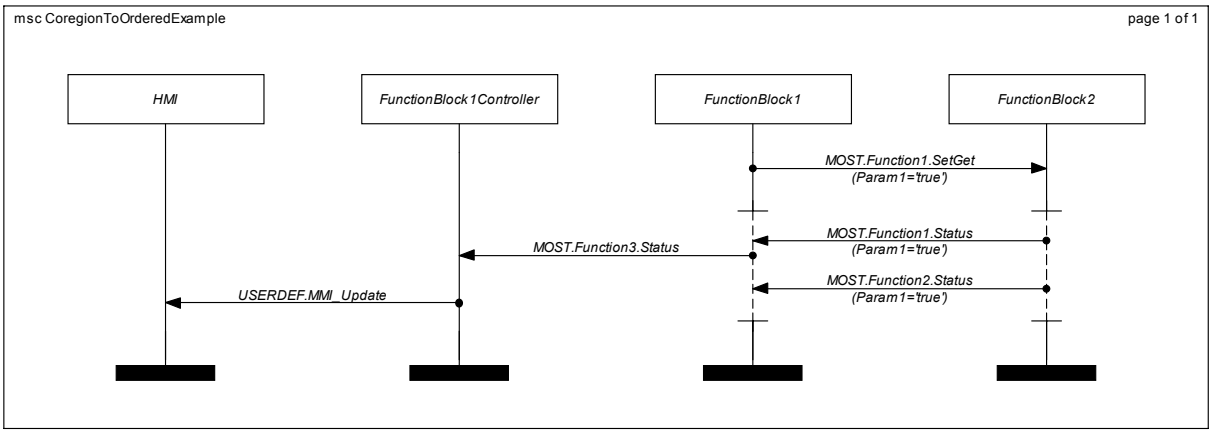
4.13 Coregion

A coregion, whose graphical representation is a vertical dashed line delimited by short horizontal lines, is part of one instance axis. All events located in the coregion are unordered.

A number of coregions on different instances can be used at the same time:



From an MSC standard point of view, even four sequences would be possible in this example if message overtaking was taken into consideration. In a MOST environment, messages will not overtake each other, so the number of potential sequences is limited to two.



This example shows how coregions can be used when communicating with instances that require ordering of messages.
The reception of MOST.Function1.SetGet in FunctionBlock2 triggers the sending of two messages in arbitrary order. Therefore, of course, FunctionBlock1 should also be able to receive them in any order. At some point before, during or after the reception FunctionBlock1 shall send MOST.Function3.Status to its controller instance. The controller instance itself requires ordering and will only notify the HMI instance after the MOST.Function3.Status message has arrived.

The MSC standard does not permit placing inline expressions or MSC references inside coregions.
In fact, only “orderable events” are allowed. Orderable events are messages and actions boxes.

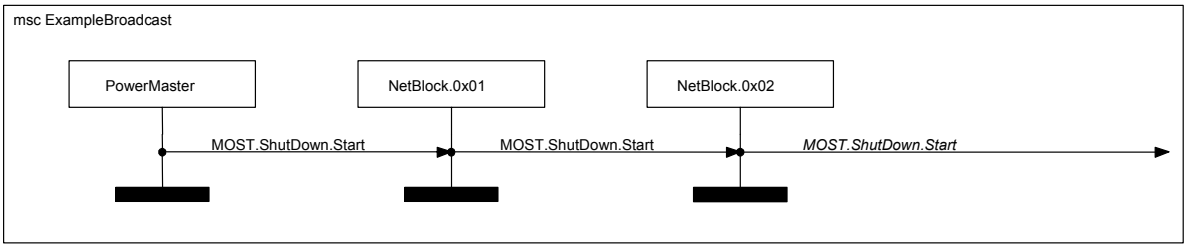
4.14 Broadcast / Groupcast

The MSC2000 standard does not provide a vocabulary for broadcast or groupcast messages, i.e. for messages with multiple receiving instances. This is, however, a crucial element in bus oriented communication protocols. In the following, a convention for the representation of broadcast/groupcast messages is described that is conforming to the MSC 2000 standard.

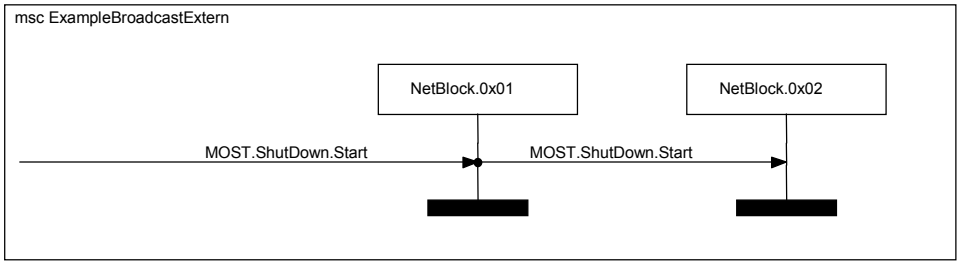
4.14.1 Recommended Approach

For exchanging message sequence charts via the standard textual representation, a special identification of broadcast and groupcast messages is necessary. For this purpose, a "note" will be employed. Optionally, also a special graphical marking can be used in the diagrams.
Ordinary messages with a single receiving instance are to be placed on different vertical positions, i.e. no two message arrows are to be placed on the same height. For broadcast or groupcast messages, however, multiple arrows will be placed on the same vertical position, so that multiple receiving instances can be shown in the diagram. Additionally, each broadcast/groupcast message will involve the environment (via a message from or to an environment gate, cp. section "4.6 Environment and Gates"), indicating that there might be additional recipients outside of the scope of the message sequence chart.

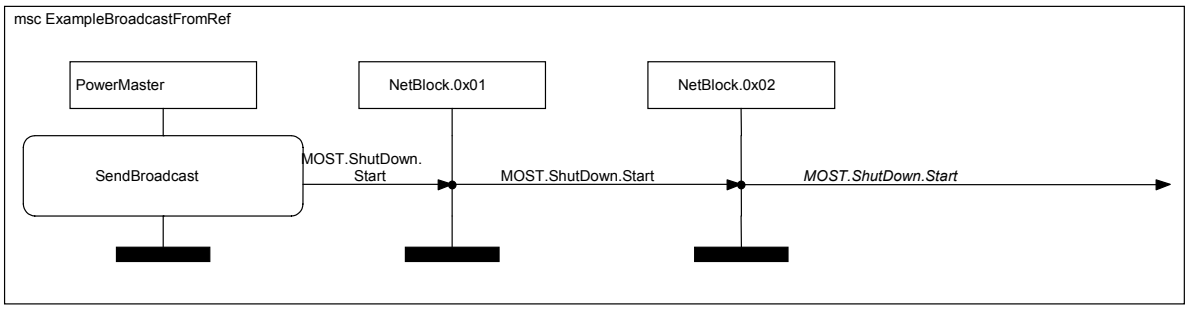
See the following diagram for an example of a broadcast message originating from the NetworkMaster and being received by two NetBlock instances:



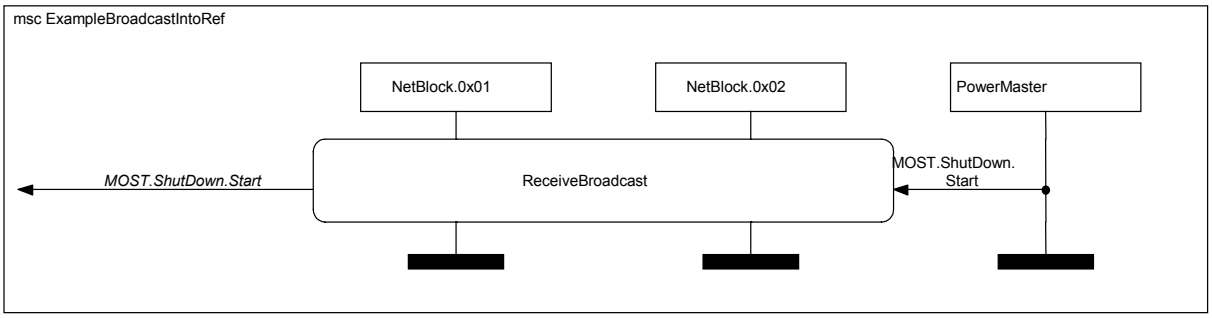
If the sender of the broadcast/groupcast message is not shown in the diagram, a message from the environment is included on the same vertical position:



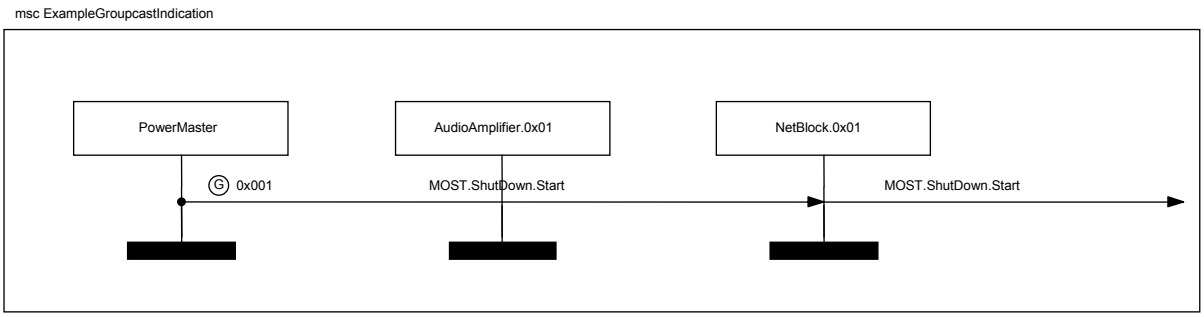
In both cases, a gate name for the message to/from the environment can be given (not shown in the example diagrams).
This convention also allows for specifying broadcast/groupcast messages into or out of referenced MSCs. When an MSC reference is associated with a subset of the instances in an MSC, a broadcast can originate in the referencing MSC and be received within the referenced MSC or vice versa. See the following examples:



This shows a broadcast message sent within a referenced MSC and received in the referencing MSC. The general mechanism of gate and environment messages is described in section "4.6 Environment and Gates". In the following example, a broadcast message originating in the referencing MSC can be received within the referenced MSC:



Optionally, a special graphical marking can be employed to explicitly mark messages as broadcast or groupcast and to differentiate between the two types. The single letter "B" or "G" respectively in a circle can be displayed above one of the message arrows. Additionally, a groupcast address can be specified for groupcast messages. Note that this is not part of the MSC2000 standard.



In the textual representation of the message sequence chart, the broadcast/groupcast message will be represented by a set of messages, one for each receiver and with a common sender. To indicate the special meaning and the grouping, a broadcast or groupcast id is assigned in a special note. See the following example fragment (this corresponds to the first example in this section, broadcast id highlighted):

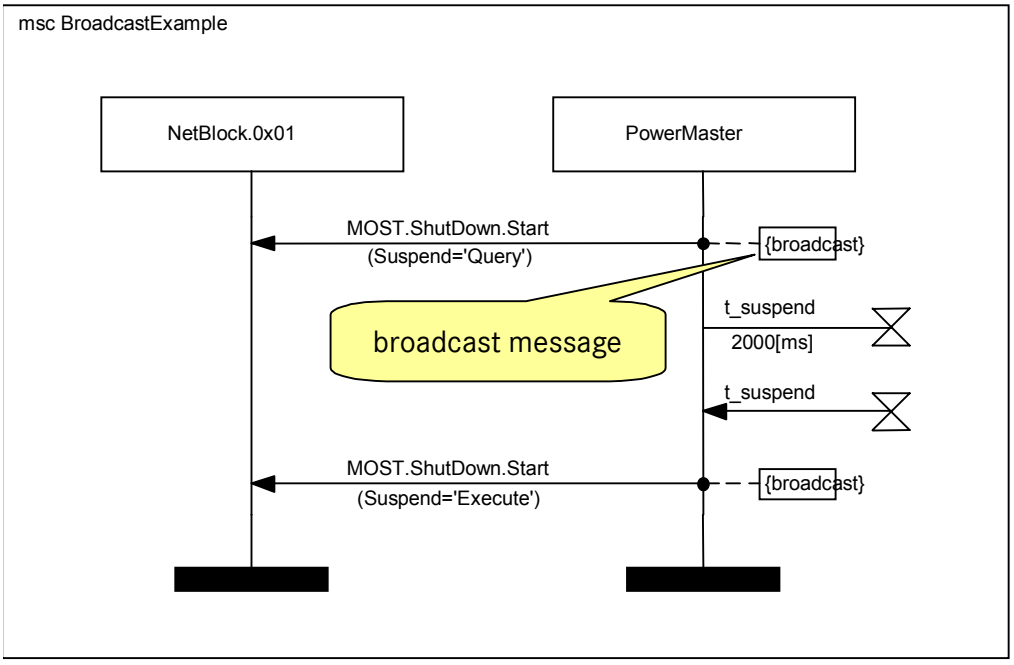
```
PowerMaster: label L0; out MOST.ShutDown.Start,1 to NetBlock.0x01
    /* BroadcastId:1 */;
NetBlock.0x01: in MOST.ShutDown.Start,1 from PowerMaster;
PowerMaster: label L1; out MOST.ShutDown.Start,2 to NetBlock.0x02
    /* BroadcastId:1 */;
NetBlock.0x02: in MOST.ShutDown.Start,2 from PowerMaster;
PowerMaster: label L2; out MOST.ShutDown.Start,3 to env
    /* BroadcastId:1 */;
```

4.14.2 Alternative Representations (not recommended)

For the sake of completeness we will mention two alternative ways of representing Broadcast/Groupcast Messages, even if we do not recommend that you use them.

4.14.2.1 Broadcast Comment

In this variation, Broadcast messages are marked with the comment “{broadcast}” (the curly brackets are an indication for the test generator to evaluate this comment. They are used to differentiate them from normal comments.)
In the example below, two broadcast messages are sent to every NetBlock instance in the system. Each broadcast message has a “{broadcast}” comment. Only one message is sent, but it is received by a number of instances. The reader of the MSC has to be familiar with MOST and “imagine” those receives.

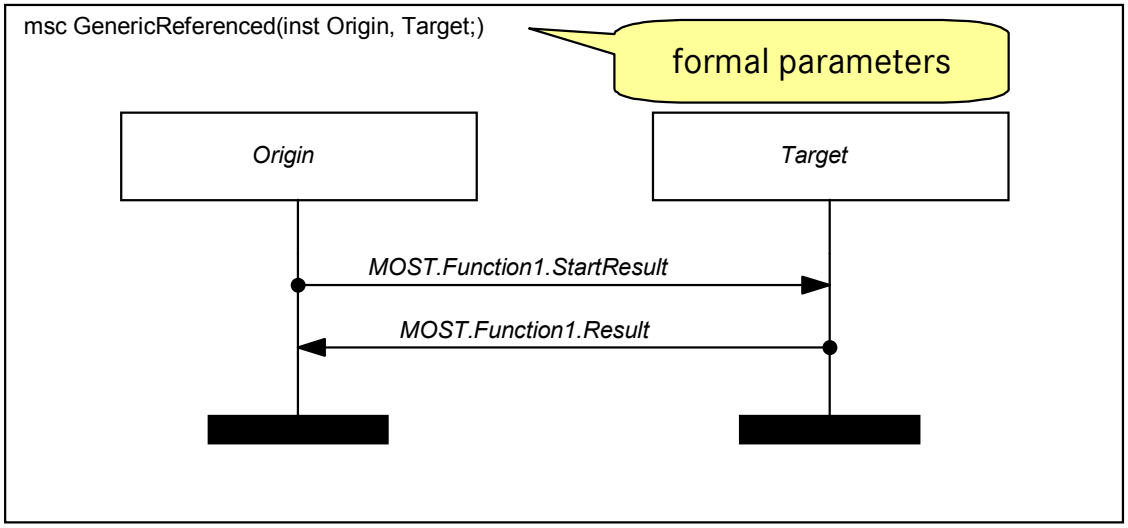


4.14.2.2 Lost and Found Messages

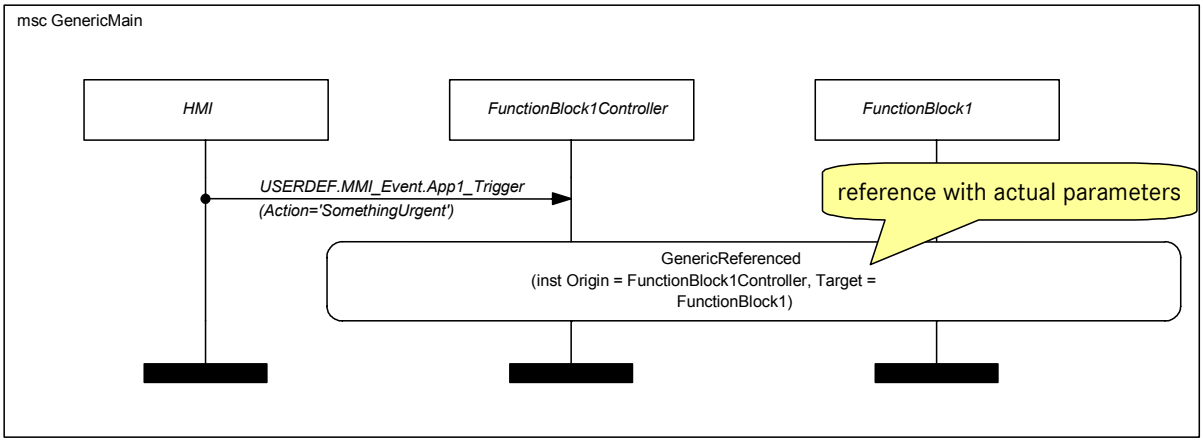
Alternatively, lost and found messages can be used to represent a broadcast/groupcast message. However, they hold the disadvantages that we mentioned when we described the concept of lost/found messages (either sender or receiver are unknown).

4.15 Generic instances

Instances in MSCs that are placeholders for concrete instances are called generic instances. An MSC that contains at least one generic instance is called a generic MSC.
Generic instances are those which appear in the list of **formal parameters**, in parentheses right behind the MSC name. In the example below, both instances are generic because both “Origin” and “Target” are contained in the formal parameters list.



If a generic MSC is referenced in another MSC, the generic instances have to be bound, i.e. it has to be defined which instance of the referencing MSC maps to which generic instance in the referenced generic MSC.
In the example below, the generic MSC from above is referenced. The binding is done by providing **actual parameters** in the reference. “FunctionBlock1Controller” is assigned to “Origin”, while “FunctionBlock1” is assigned to “Target”.



5 Appendix A: Extension to MSC2000 (Named Parameters)

5.1 Rationale / Requirements

When employing Message Sequence Charts for the specification, definition, and description of communication protocols with complex parameter types, the simple message format of the MSC2000 Standard and commonly used data languages reveal their shortcomings. Especially when looking at MOST communications, the following requirements are identified:

- naming of parameters in messages (e. g. SourceNr, Data, ErrorInfo, etc.)
- representation of complex types (arrays, records, arrays of records etc.)
- different parameter formats for the same message (e. g. complete array of records, single record, single entry etc.)

Therefore, by common agreement between DaimlerChrysler, BMW, and a number of tool vendors and software suppliers, an extension of the MSC2000 Standard was defined. A respective proposal will be submitted to ITU for a future release of the MSC Standard.
In the following sections a simple, backward compatible extension is described that introduces optional parameter names. This is complemented by a convention for using parameter expressions for complex types - with the advantage of maintaining static message signatures.

5.2 Named Message Parameters

In the MSC2000 Standard, message parameters can be defined solely "by position", i.e. parameter values can be specified - and the order determines the assignment to declared parameters, e.g.

```
MOST.Allocate.Result('1', '10', 'Channel1')
```

In the extended format, optional definitions of parameter names are allowed, e. g.

```
MOST.Allocate.Result(SourceNr='1', SrcDelay='10', ChannelList='Channel1')
```

with the parameter names defined in the MOST Function Catalog. When parameter names are specified, the order of the parameters is not fixed - but it is not permissible to mix positional and named specifications within the same message.

Thus, partial specifications become possible by omitting irrelevant parameters, e. g.

```
MOST.Allocate.Result(SourceNr='1', ChannelList='Channel1')
```

Optionally, the omission can be indicated in the diagram by ellipses ("..."), but not in the MPR file:

```
MOST.Allocate.Result(SourceNr='1', ..., ChannelList='Channel1')
```

Alternatively, a parameter name can be specified without assigning a value by using wildcards (default: "_"), e. g.

```
MOST.Allocate.Result(SourceNr='1', SrcDelay=_, ChannelList='Channel1')
```

Parameters, which are omitted, are treated as if wildcards had been provided.

5.3 Hierarchical Representation of Complex Values

In MOST Function Catalogs, complex parameter types are defined (e. g. arrays of records). When representing these in messages, a hierarchical format is employed. An array of records is specified by using a pair of parameters "Pos" and "Data". The latter of these contains the array of records, or part of it, depending on the value of "Pos". Example:

```
MOST.ATPresetList1.Status(Pos={x='0', y='0'},
        Data={PresetSelection[0]=_, Sendername[0]=_, SendernameInfo[0]=_,
            PTY[0]=_, TpTa[0]=_, PI[0]=_, PresetSelection[1]=_,
Sendername[1]=_,
            SendernameInfo[1]=_, PTY[1]=_, TpTa[1]=_, PI[1]=_, ...})
```

Curly brackets ("{", "}") are used to delimit the value of a complex parameter - this constitutes a legal <expression string> in terms of the MSC2000 Standard. Within the curly brackets, the same named parameter format is employed, i.e. parameter names, omissions, wildcards and ellipses (in graphical representation only!) are used. Note the convention for indicating array members (indices in square brackets) and the usage of record member names. The record is not named, so the generic (sub-)parameter name is

```
RecordMemberName[ArrayIndex]
```

In the above example, the parameter "Pos" has two components, x and y. Setting both to 0 requires the complete array of records to be transmitted in "Data". But different settings of "Pos" specify that "Data" contains only parts of the array of records. One advantage of the chosen approach is that the signature of the overall message ("Pos"/"Data") is still static.

Stream parameters are not represented in hierarchical form, as there are many different uses for this kind of parameters. Instead, the parameter value is treated as a string that is to be parsed by the target application in the context of the specific message type. E.g.

```
MOST.Allocate.Result(ChannelList='Channel1, Channel2, ...')
```

5.4 Named MSC/Reference Parameters

The MSC2000 allows for the declaration of MSC parameters (instance, message, data, and timer parameters) and their assignment in MSC references. Both declaration and assignment also profit from the use of parameter names.

Therefore, the format for named message parameters is applied here analogously. E.g.

```
msc Handshake(inst Source, Sink)
```

defines two instance parameters (“Source”, “Sink”) for MSC “Handshake”. A reference to “Handshake” can assign actual values (instances from the referring MSC) to these parameters by using

```
reference Handshake(inst Source = AmFmTuner, Sink = AudioAmplifier)
```

In the diagram, only the reference expression

```
Handshake(inst Source = AmFmTuner, Sink = AudioAmplifier)
```

will be shown within the MSC reference symbol.

The declaration and assignment of message, data, and timer parameters is analogous (with the respective keywords "msg", "variables", and "timer").

6 Appendix B: Tool Requirements

A tool is required to edit MSCs. The most frugal way to view, create and edit MSCs is to use a text editor, because MSCs are based on a textual representation with a defined grammar.
For good reasons, most people will shy away from that alternative and rather utilize a more sophisticated instrument – a graphical editor.
In the following, we will outline the basic requirements and characteristics of such a tool. We will call it "MSC editor".

6.1 The MSC Editor

The MSC editor is the central tool for handling MSCs. Its main features shall support the composition of MSCs, while additional features support the seamless integration into tool chains and development processes that are common in the automotive telematics/infotainment domain.

The MSC editor shall be compliant to the Z.120 recommendation, both when reading and writing MSC files. At the same time it shall support required extensions, e.g. "named parameters".
The composition of MSCs typically is performed by software architects and engineers who specify requirements for applications and devices. Those are usually familiar with flowcharts and UML diagrams. Therefore, the MSC editor shall work similar to commonly used tools for drawing flowcharts or creating UML diagrams.

6.1.1 Main Features of the MSC Editor

The MSC editor shall support an intuitive graphical composition of MSCs (MSC-GR) and the modification of existing MSCs. It shall support both HMSCs (High-Level MSCs) and Basic MSCs.
The graphical representation shall be based on a layout engine which uses default settings to automatically display the MSCs in a way that is convenient for a majority of the users, at the same time allowing experienced users to modify those settings to meet their special needs.

The MSCs shall be stored in their textual representation (MSC-PR).
The MSC editor shall support the structuring of MSCs by allowing to group MSCs into scenarios.
Further information about scenarios and individual MSCs within a scenario can be added and displayed, e.g. by the use of MSC comments.

The MSC editor shall offer the user WWW-browser-like functionality to follow references and navigate back and forth between references. It shall have drag&drop capability to let the user move parts of MSCs or entire MSCs.

6.1.2 Additional Features of the MSC Editor

Versioning of MSCs is done on the scenario-level (scenarios are mapped to files). The MSC editor shall support the embedding of version control tags for commonly used version control systems, e.g. by adding tags to the very top of each *mpr-file (MSC-PR) under version control. These tags are visible in the textual representation and may also be displayed in the MSC editor's GUI.

MSCs usually become part of the requirements specification. Therefore, the tool shall support the connection to one or more requirements management systems, which requires a certain degree of automation. The user shall be able to view and modify MSCs from their requirements management system, depending on the particular development process.

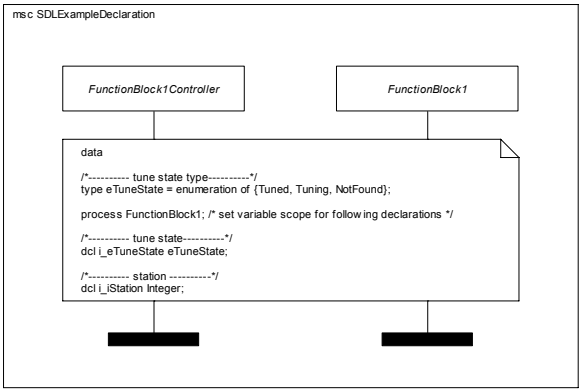
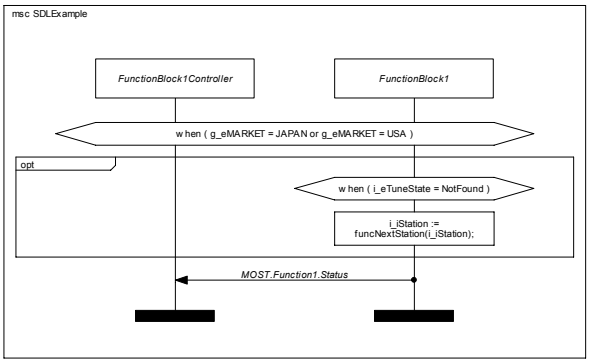
In the MOST domain, the MOST FunctionCatalog is commonly used. The MSC editor shall be able to parse the XML representation of the MOST FunctionCatalog, and offer the user syntactical help during the specification of MOST communication. The CAN protocol is another commonly used communication format. It shall be supported by the MSC editor in a way that is similar to the handling of the MOST FunctionCatalog, wherever applicable.
For different car models, the user shall be able to select different catalogs or databases.

7 Appendix C: Data Language

In the MSC Cookbook, SDL was used in many examples that involved data. Here we will introduce you to the extended subset of the SDL language that we have used in a number of MSCs.

7.1 Constructs

We used SDL data in MSC conditions, actions and time constraints. Data declarations are placed inside MSC text symbols. (The MSC standard expects them in the document head under the “data” section.)

Declaration	Usage
	

7.1.1 Conditions

Conditions may include

- Variables of Boolean, Enum, Integer, Real and String types
- Boolean expressions

7.1.2 Actions

Actions include

- Assignments to Variables
- Function Calls

7.1.3 Time Constraints

Time constraints may consist of

- Variables of Real type

7.1.4 Text Symbols

Text symbols are used for

- Data Declarations

7.2 Data Types

We mainly rely on Boolean, Enum, Integer, Real and String types. This chapter contains their characteristics and the supported operations.

Declaration of data

```
data
<scope identifier>;
dcl <variable_name1> <variable_type>;
dcl <variable_name2>, <variable_name3> <variable_type>;
```

7.2.1 Boolean Type

This type can have two literal values, False and True.

Operators

Operator	Description
not	inverts the value.
and	if both parameters are False the result is False, else it is True.
or	if both parameters are False then the result is False, else it is True
xor	if parameters are different then the result is True, else it is False
=	equal
/=	not equal

7.2.2 Enum type

This type can be assigned to those constants that have been provided during its declaration.

Enum type declaration example

```
data
type eMarket = enumeration of {ECE, USA, JAPAN};
```

Operators

Operator	Description
=	equal
/=	not equal

7.2.3 Integer type

The integer type is used to represent mathematical integers. The literal values of the Integer type are the usual unsigned decimal numbers. Negative numbers are represented using the unary "-" operator.

Operators

Operator	Description
=	Equal
/=	not equal
+	Addition
-	subtraction or "unary minus"
*	Multiplication
/	integer division
mod	modulus in integer division.
rem	remainder in integer division.
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

7.2.4 Real Type

The real type is used to represent mathematical rational numbers.

Operators

Operator	Description
=	Equal
/=	not equal
+	Addition
-	subtraction or "unary minus"
*	Multiplication
/	Division
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

7.2.5 String Type

The string type is used to represent sequences of alphanumerical characters.

Operators

Operator	Description
=	Equal
/=	not equal

7.3 Examples

In the following, we give a few examples of data language use.

7.3.1 Declaration

Declarations are introduced by the keyword “data“.

```
data
global;
constant g_eMARKET eMarket := 'USA';
dcl g_iTemperature Integer;
dcl g_bCANAvailable Boolean;

process Telephone;
dcl i_sTelephoneNumber String;

process AmFmTuner;
dcl i_rTunedFrequency Real;
```

In this example, one global constant and two global variables are declared. The global constant that represents the market is initialized with the value ‘USA’.
In addition, two instance-local variables are declared: A telephone number string that belongs to the Telephone instance and a Real number value for the AmFmTuner Frequency.

7.3.2 Usage of Variables and Functions in MSCs

Variables can be assigned, compared and used in function calls. The optional SDL keyword “task” can be used to determine that an action is formal.

```
task l_iLoopCounter := l_iLoopCounter + 1;

task i_sCurrentStations := GetStationString(i_rTunedFrequency);
```

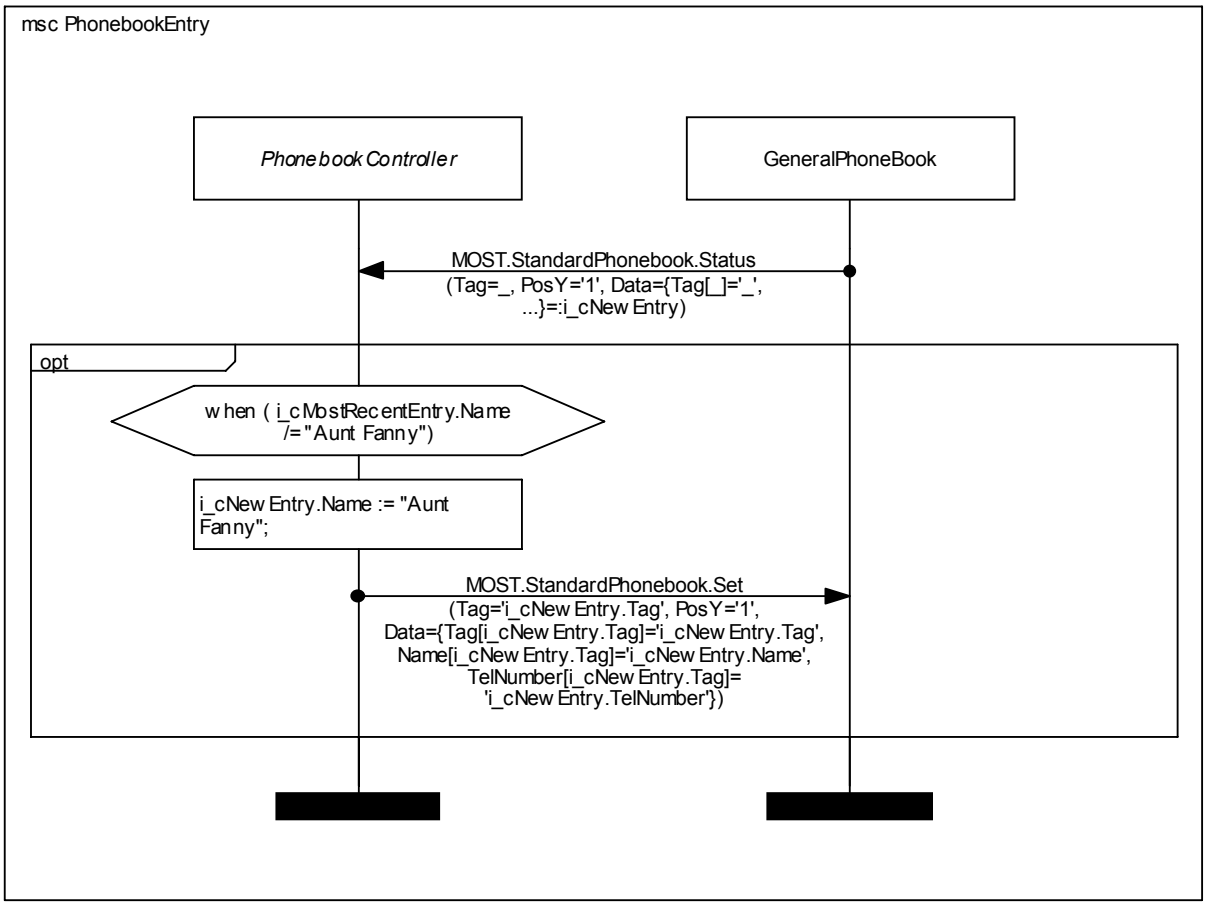
7.3.3 Using MOST Catalog Types

A common problem when working with MSCs is properly using data items that are specified in MOST FunctionCatalogs.
The complex compositions of basic types in records and arrays used in the XML representation of MOST FunctionCatalogs are not named. That makes it hard to specify MSCs that use variables for MOST message parameters. For example, how can we create a variable that represents a phonebook entry with tag, name and telephone number?
We tackle the problem by making an assumption: Structural identity is equivalent to type identity. Whenever only the names of types differ, the types shall be interchangeable regardless. This arrangement enables us to declare user-defined types and use them in place of MOST types.

```
type PhonebookEntry=record of
{
    Tag integer;
    Name string;
    TelNumber integer;
};

process PhonebookController;
dcl i_cNewEntry PhonebookEntry;
```

In this example, you see the declaration of a record that represents a phonebook entry. A variable of that user-defined type is declared. In the MSC below, we rely on this declaration and variable definition.



In the PhonebookEntry example, we assume that Aunt Fanny's number was the last one dialed. The user now wants to store the number under the entry "Aunt Fanny".

The PhonebookController receives the most recently made entry to the phone book. If the name is already set to "Aunt Fanny", nothing happens; otherwise the "Name" member is modified.
This example shows how implicitly known MOST types and user-defined data types can be used in the same context to address a StandardPhonebook entry.

Notes:

Notes: