

ETSI EN 301 234 V2.1.1 (2006-05)

European Standard (Telecommunications series)

Digital Audio Broadcasting (DAB); Multimedia Object Transfer (MOT) protocol

European Broadcasting Union



Union Européenne de Radio-Télévision

EBU-UER

DAB
Digital Audio Broadcasting



Reference

REN/JTC-DAB-38A

Keywords

audio, broadcasting, DAB, digital, multimedia,
protocol

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Important notice

Individual copies of the present document can be downloaded from:

<http://www.etsi.org>

The present document may be made available in more than one electronic version or in print. In any case of existing or perceived difference in contents between such versions, the reference version is the Portable Document Format (PDF). In case of dispute, the reference shall be the printing on ETSI printers of the PDF version kept on a specific network drive within ETSI Secretariat.

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at

<http://portal.etsi.org/tb/status/status.asp>

If you find errors in the present document, please send your comment to one of the following services:

http://portal.etsi.org/chaicor/ETSI_support.asp

Copyright Notification

No part may be reproduced except as authorized by written permission.
The copyright and the foregoing restriction extend to reproduction in all media.

© European Telecommunications Standards Institute 2006.

© European Broadcasting Union 2006.

All rights reserved.

DECT™, **PLUGTESTS™** and **UMTS™** are Trade Marks of ETSI registered for the benefit of its Members.
TIPHON™ and the **TIPHON logo** are Trade Marks currently being registered by ETSI for the benefit of its Members.
3GPP™ is a Trade Mark of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners.

Contents

Intellectual Property Rights	6
Foreword.....	6
Introduction	7
1 Scope	8
2 References	8
3 Definitions and abbreviations.....	9
3.1 Definitions	9
3.2 Abbreviations	11
4 General description of the MOT protocol	12
4.1 Requirements of Multimedia services	12
4.2 Problems MOT solves	12
4.3 Receiver architecture reference model	13
5 Structural description	14
5.1 Segmentation of MOT entities	15
5.1.1 Segmentation header.....	17
5.1.2 Segmentation of the MOT body	17
5.1.3 Segmentation of the MOT directory	17
5.1.4 Segmentation of the MOT header.....	18
5.2 Transporting MOT segments - network level.....	18
5.2.1 Packet mode.....	18
5.2.2 X-PAD	18
5.3 Transmission Mechanisms	20
5.3.1 Single object transmission (MOT header mode)	20
5.3.1.1 Repetition on object level.....	20
5.3.1.2 Repetition of MSC data groups/MOT segments	21
5.3.1.3 Repeated transmission of header information (combined with repetition on object level)	21
5.3.2 Multiple object transmissions (MOT directory mode).....	22
5.3.2.1 Interleaving MOT entities in one MOT stream.....	23
6 MOT header information.....	24
6.1 Header core	24
6.2 Header extension	25
6.2.1 Future expansion of the parameter data field.....	26
6.2.2 Parameters of the header extension for MOT header mode and MOT directory mode	27
6.2.2.1 MOT Basic transport parameters	27
6.2.2.1.1 ContentName	27
6.2.2.1.2 MimeType	27
6.2.2.1.3 CompressionType.....	28
6.2.3 Parameters of the header extension for MOT directory mode only	28
6.2.3.1 MOT caching support parameters	28
6.2.3.1.1 Expiration	28
6.2.3.1.2 PermitOutdatedVersions.....	30
6.2.3.1.3 UniqueBodyVersion	30
6.2.3.1.4 Priority.....	30
6.2.3.1.5 RetransmissionDistance.....	30
6.2.3.2 MOT conditional access parameters	31
6.2.3.2.1 CAInfo.....	31
6.2.3.2.2 CAReplacementObject.....	31
6.2.3.3 MOT profile identification.....	31
6.2.3.3.1 ProfileSubset	32
6.2.4 Coding of parameters.....	32
6.2.4.1 Coding of time parameters	32
6.3 List of all MOT parameters in the MOT header extension.....	32

7	MOT transport modes	34
7.1	MOT header mode.....	35
7.1.1	New object/object update.....	35
7.1.2	Management of TransportIds.....	36
7.1.3	Updating header information/triggering objects	36
7.2	MOT directory mode.....	37
7.2.1	Introduction.....	37
7.2.2	Assembly of MOT bodies and MOT directory	37
7.2.3	MOT directory coding	37
7.2.4	List of all MOT parameters in the MOT directory extension	39
7.2.4.1	SortedHeaderInformation.....	39
7.2.4.2	DefaultPermitOutdatedVersions	40
7.2.4.3	DefaultExpiration.....	40
7.2.5	Segment size of the MOT directory.....	40
7.2.6	Identification of the MOT directory.....	41
7.2.7	Use of the MOT directory mode.....	41
7.2.7.1	Segment reception order	41
7.2.7.2	Service acquisition	41
7.2.7.3	Version control.....	41
7.2.7.4	Allocation of TransportIds	42
7.2.7.5	Prioritizing objects within the data carousel	42
7.2.7.6	Managing updates to the data carousel.....	42
7.2.7.7	MOT decoder behaviour in case no data is received for a long time	42
7.2.8	MOT directory compression	43
8	MOT functionality.....	44
8.1	MOT caching support (MOT directory mode only)	44
8.1.1	Object reassembly.....	46
8.1.2	Object validity	46
8.1.2.1	MOT expire time handling.....	48
8.1.2.2	Unique MOT body version	49
8.1.2.3	Temporarily using outdated MOT bodies	50
8.1.3	Object management	51
8.2	Transfer of directory structures using MOT.....	52
Annex A (normative): Comparing ContentNames.....		54
Annex B (informative): User application definitions and MOT		55
Annex C (informative): Model of an MOT decoder and its interfaces		57
C.1	Network level.....	59
C.2	MSC Data group level.....	60
C.3	Segmentation and object level.....	60
C.3.1	General description of the MOT decoder	60
C.3.2	The reassembly unit.....	60
C.3.2.1	MOT directory mode	60
C.3.2.2	MOT header mode	61
C.3.2.3	Segmentation of MOT bodies.....	61
C.3.3	The object management unit	61
C.3.3.1	MOT directory mode	61
C.3.3.2	MOT header mode	62
C.3.4	Advanced MOT reassembly units	62
C.3.4.1	Collecting MOT body segments whose TransportId is not described in the MOT directory	62
C.3.4.1.1	Start-up of the MOT directory mode decoder	63
C.3.4.1.2	Updates to the MOT directory	63
C.3.4.1.3	Collecting MOT body segments	64
C.3.4.2	MOT caching support: relative expire times (MOT parameters Expiration and DefaultExpiration).....	65
C.3.4.3	Acquiring both compressed and uncompressed MOT directories	65
C.3.5	Advanced MOT object management.....	68
C.3.5.1	MOT directory mode	68

C.3.5.1.1	Support of MOT parameters DefaultPermitOutdatedVersions and PermitOutdatedVersions	70
C.4	User application level.....	71
Annex D (informative):	MOT decoding in MOT directory mode (example).....	72
Annex E (informative):	Example for evaluation of relative expire times (MOT parameters Expiration and DefaultExpiration)	73
Annex F (informative):	Managing changes to the MOT data carousel.....	75
F.1	General principle	75
F.2	Advanced approach.....	75
Annex G (informative):	Implementation tips for "Transfer of directory structures via MOT" (receiver side)	77
Annex H (informative):	Bibliography.....	78
History	79

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: "*Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards*", which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (<http://webapp.etsi.org/IPR/home.asp>).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Foreword

This European Standard (Telecommunications series) has been produced by Joint Technical Committee (JTC) Broadcast of the European Broadcasting Union (EBU), Comité Européen de Normalisation ELECtrotechnique (CENELEC) and the European Telecommunications Standards Institute (ETSI).

NOTE 1: The EBU/ETSI JTC Broadcast was established in 1990 to co-ordinate the drafting of standards in the specific field of broadcasting and related fields. Since 1995 the JTC Broadcast became a tripartite body by including in the Memorandum of Understanding also CENELEC, which is responsible for the standardization of radio and television receivers. The EBU is a professional association of broadcasting organizations whose work includes the co-ordination of its members' activities in the technical, legal, programme-making and programme-exchange domains. The EBU has active members in about 60 countries in the European broadcasting area; its headquarters is in Geneva.

European Broadcasting Union
CH-1218 GRAND SACONNEX (Geneva)
Switzerland
Tel: +41 22 717 21 11
Fax: +41 22 717 24 81

The Eureka Project 147 was established in 1987, with funding from the European Commission, to develop a system for the broadcasting of audio and data to fixed, portable or mobile receivers. Their work resulted in the publication of European Standard, EN 300 401 [1], for DAB (see note 2) which now has worldwide acceptance. The members of the Eureka Project 147 are drawn from broadcasting organizations and telecommunication providers together with companies from the professional and consumer electronics industry.

NOTE 2: DAB is a registered trademark owned by one of the Eureka Project 147 partners.

National transposition dates	
Date of adoption of this EN:	19 May 2006
Date of latest announcement of this EN (doa):	31 August 2006
Date of latest publication of new National Standard or endorsement of this EN (dop/e):	28 February 2007
Date of withdrawal of any conflicting National Standard (dow):	28 February 2007

Introduction

The present document is a complete revision of V1.2.1. It does not change the former MOT specifications but tries to give a much more extensive and understandable description of what MOT is, how it works and how an MOT decoder can be implemented. The present document concentrates on describing the transport related issues; all user application specific issues will now be handled by the user applications specifications.

MOT considers itself the multimedia transport protocol that provides the necessary tools to carry all kind of multimedia data. It is the user application to decide which of the tools provided by MOT it uses and the user applications might also extend or restrict some functionality that is described in the present document.

Some general and transport related MOT parameters (`MimeType`, CA related parameters, `CompressionType`, `ProfileSubset`) were removed from the MOT broadcast web site specification [6] and added to the present document.

User application specific parameters (`TriggerTime`, `Label`) are no longer explained in the present document and will be described in user application specifications (MOT broadcast website [6]; MOT Slide Show [5]).

A clear indication of the differences between MOT header mode and MOT directory mode is given.

Backwards compatible extensions reduce the footprint of the MOT decoder and permit much better user behaviour during changes to a data carousel. MOT directory compression permits better use of the broadcast channel.

A detailed description of a model of an MOT decoder and its interfaces is given in annex C. This should help implementing a fully standard compliant and efficient MOT decoder. Additional clauses describe features that an advanced MOT decoder should use to further improve the performance of the MOT decoder and to enhance the user experience.

The present document contains identifier fields that require values to be registered. Registered value lists associated with data broadcasting specifications for DAB are maintained by the WorldDAB Information and Registration Centre (WIRC). Since the lists and tables contained within the present document might be outdated, please refer to the most recent versions of TS 101 756 [7]. The present document also describes the procedures for registering values in an existing table as well as registering new tables.

1 Scope

The present document specifies a transmission protocol, which allows to broadcast various kinds of data using the Digital Audio Broadcasting (DAB) system. It is tailored to the needs of Multimedia services and the specific constraints given by the broadcasting characteristics of the DAB system. After reception this data can be processed and presented to the user.

The present document defines the transport specific encoding for data types not specified in EN 300 401 [1] according to the transport mechanisms provided by DAB. It allows a flexible utilization of the data channels incorporated in the DAB system, as well as methods to manage and maintain a reliable transmission in a uni-directional broadcast environment. Provisions are also made for the creation and presentation of advanced Multimedia services using formats such as Hyper Text Markup Language (HTML) (see RFC 2854 [2]) or Multimedia and Hypermedia information coding Experts Group (MHEG) (see ISO/IEC 13522 [3]).

The present document describes the core transport protocol. Subsequent parts or revisions of the present document will describe backwards compatible extensions.

Aspects related to the further decoding and processing of the data objects carried are outside the scope of the present document. Hardware implementation considerations are not covered.

2 References

The following documents contain provisions which, through reference in this text, constitute provisions of the present document.

- References are either specific (identified by date of publication and/or edition number or version number) or non-specific.
- For a specific reference, subsequent revisions do not apply.
- For a non-specific reference, the latest version applies.

Referenced documents which are not found to be publicly available in the expected location might be found at <http://docbox.etsi.org/Reference>.

- [1] ETSI EN 300 401: "Radio Broadcasting Systems; Digital Audio Broadcasting (DAB) to mobile, portable and fixed receivers".
- [2] IETF RFC 2854: "The 'text/html' Media Type".
- [3] ISO/IEC 13522 (all parts): "Information technology - Coding of multimedia and hypermedia information".
- [4] IETF RFC 2045 to 2049: "Multipurpose Internet Mail Extensions (MIME)".
- [5] ETSI TS 101 499: "Digital Audio Broadcasting (DAB); MOT Slide Show; User Application Specification".
- [6] ETSI TS 101 498-1: "Digital Audio Broadcasting (DAB); Broadcast website; Part 1: User application specification".
- [7] ETSI TS 101 756: "Digital Audio Broadcasting (DAB); Registered Tables".
- [8] ETSI TS 102 367: "Digital Audio Broadcasting (DAB); Conditional access".
- [9] ETSI TR 101 496-2: "Digital Audio Broadcasting (DAB); Guidelines and rules for implementation and operation; Part 2: System features".
- [10] ISO/IEC 8859-1: "Information technology - 8-bit single-byte coded graphic character sets - Part 1: Latin alphabet No. 1".
- [11] ISO/IEC 10646: "Information technology - Universal Multiple-Octet Coded Character Set (UCS)".

3 Definitions and abbreviations

3.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

byte ordering: all numeric values using more than one byte have to be coded in Big Endian Format (most significant byte first)

NOTE: In all schematics the bits are ordered with the most significant bit of a byte ("b7") at the left end and least significant bit ("b0") at the right end of the drawing.

Conditional Access (CA): mechanism by which user access to service components can be restricted

content provider: provides data for a user application instance

NOTE: It is his responsibility to provide the data according to the user application standard and to transmit it according to the transport protocol used by the user application. The provided data is transmitted in a data channel of a DAB data or programme service.

DAB receiver: Multimedia Object Transfer (MOT) specific definition of a DAB receiver includes decoding of the DAB signal and resolving the multiplex structure of the main service channel

data carousel: delivery system that allows the broadcast component of a user application to present a set of distinct objects to a user application decoder by cyclically repeating the contents of the data carousel

NOTE: For some user applications the data carousel may complete only a few or a single cycle.

data channels: data channels in DAB (packet mode, X-PAD) provide the functionality on the transport layer in order to convey the objects

data decoder: data decoder processes the MOT data stream and applies both packet mode/X-PAD specific decoding and then MOT decoding

directory core: directory core contains basic information describing the data carousel (e.g. number of objects, data carousel period)

NOTE: The directory core does not describe individual objects.

directory extension: directory extension contains additional information about the data carousel. The directory extension does not describe individual objects

ensemble: transmitted signal, comprising a set of regularly and closely-spaced orthogonal carriers

NOTE: The ensemble is the entity which is received and processed. In general, it contains programme and data services.

Fast Information Channel (FIC): part of the transmission frame, comprising the Fast Information Blocks (FIB), which contains the multiplex configuration information together with optional service information and data service components

header core: header core contains information about the size and the content type of the object, so that the receiver can determine whether it has system resources to decode and present the object or not

header extension: header extension includes additional information about the object

header information: header information consists of the header core and the header extension and describes one MOT body

NOTE: The header information can be sent in an MOT header (the MOT header describes one single MOT body) or an MOT directory (the MOT directory describes all MOT bodies within the data carousel).

MOT body: carries any kind of data of finite length. The MOT body is described by the MOT header information

MOT data service: data service comprises information that is intended to be presented to a user, i.e. text, pictures, video or audio sequences

NOTE: A user application decoder is required to gain access to the data. This might be a viewer which decodes text and pictures and displays them on a screen. In terms of MOT a data services consists of one or an ordered collection of several objects. It is not in the scope of MOT to deal with the content of the object, but to carry information to support both presentation and handling of these objects.

MOT directory: within a data carousel the MOT directory contains a complete description of the content of the data carousel

NOTE: It includes the MOT header information of all objects within the data carousel.

MOT entity: single MOT body, a single MOT directory or a single MOT header

MOT header: this MOT entity contains the header information that describes one single MOT body

MOT object: used to transfer data in DAB. The object consists of header information and an MOT body carrying the payload

MOT parameter: MOT parameter provides information about an MOT object or about the data carousel as a whole

NOTE: MOT parameters describing one single MOT object are carried in the header information of the MOT object. MOT parameters describing a data carousel are only available in MOT directory mode and they are carried in the MOT directory extension. MOT parameters can be transport specific (MOT transport parameters) or user application specific (MOT user application specific parameters). The present document defines transport specific MOT parameters. The user application specific MOT parameters are defined in user application standards.

MOT segment: all MOT entities are split into MOT segments for transmission

NOTE: The MOT segments are then mapped to MSC data groups and inserted into a DAB packet mode subchannel or into the X-PAD channel of an audio service component.

MOT stream: MOT stream comprises all data for one user application instance

NOTE: One stream of MOT objects is transferred in an individual service component (packet mode) or as part of the X-PAD of an audio service component. Several MOT entities might be conveyed in parallel by interleaving. Note that within one packet mode subchannel or one X-PAD channel there might be multiple MOT streams carried in parallel.

Main Service Channel (MSC): channel which occupies the major part of the transmission frame and which carries all the digital audio service components, together with possible supporting and additional data service components

MSC data group: package of data for carrying one segment of an MOT object

NOTE: The MSC data group can be carried in a packet mode subchannel or in the Extended Programme Associated Data (X-PAD) part of an audio subchannel.

packet mode: mode of data transmission in which data are carried in addressable blocks called packets

NOTE: Packets are used to convey MSC data groups within a sub-channel. The packet mode carries the load in packets of a certain size, separating different streams of packets by specific addresses. Error detection and repetition are already covered by packet mode and thus allow a reliable and flexible data transmission.

Programme Associated Data (PAD): information that is related to the audio data in terms of content and synchronization

NOTE: The PAD field is located at the end of the DAB audio frame.

service: user-selectable output which can be either a programme service or a data service

service component: part of a service which carries either audio (including PAD) or data

NOTE: The service components of a given service are linked together by the Multiplex Configuration Information (MCI). Each service component is carried either in a sub-channel or in the Fast Information Data Channel (FIDC).

service label: alphanumeric characters associated with a particular service and intended for display in a receiver

TransportId: this 16-bit field shall uniquely identify one data object (body and header information) from a stream of such objects

NOTE: It shall be used to indicate the object to which the information carried in the segment belongs or relates. It is valid only during transport of the object.

user application: data application defined in a separate standard and fed with data via DAB

NOTE: A user application using MOT can be carried in a packet mode subchannel or in X-PAD. The stream of MOT objects belonging to a user application instance is called an "MOT stream", see above.

X-PAD (eXtended Programme Associated Data): extended part of the PAD carried towards the end of the DAB audio frame, immediately before the scale factor Cyclic Redundancy Check (CRC)

NOTE: It is used to transport information together with an audio stream which is related or synchronized to the X-PAD. No provisions for error detection are included in X-PAD so that additional protocols are required for some user applications.

3.2 Abbreviations

For the purposes of the present document, the following abbreviations apply:

CA	Conditional Access
CRC	Cyclic Redundancy Check
DAB	Digital Audio Broadcasting
ECM	Entitlement Checking Message (Conditional Access related)
EMM	Entitlement Management Message (Conditional Access related)
FFT	Fast Fourier Transform
FIB	Fast Information Block
FIC	Fast Information Channel
FIDC	Fast Information Data Channel
HF	High Frequency
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
JPEG	Joint Photographic Experts Group
MCI	Multiplex Configuration Information
MHEG	Multimedia and Hypermedia information coding Experts Group
MIME	Multipurpose Internet Mail Extensions
MJD	Modified Julian Date
MOT	Multimedia Object Transfer
MSC	Main Service Channel
PAD	Programme Associated Data
PLI	Parameter Length Indicator
Rfa	Reserved for future addition

NOTE: See TR 101 496-2 [9] (Guidelines DAB).

Rfu Reserved for future use

NOTE: See TR 101 496-2 [9].

UTC	Universal Time Co-ordinated
WIRC	WorldDAB Information and Registration Centre
X-PAD	eXtended Programme Associated Data

4 General description of the MOT protocol

4.1 Requirements of Multimedia services

Multimedia in general can be referred to as information and its presentation in various formats (visible, audible, etc.) and forms (text, pictures, video, etc.). The material is often structured and packaged into a number of containers or files which shall be either completely available before the presentation or are delivered on request of the user.

Multimedia services require to control the presentation (e.g. the arrangement of visible information on a screen) and therefore direct access to both hardware and software resources of the receiver/terminal is essential. The appropriate time shall also be considered for the presentation. Thus it is required to synchronize the various elements (e.g. video together with the sound), i.e. some kind of a runtime environment is necessary.

4.2 Problems MOT solves

MOT is a transport protocol for the transmission of Multimedia content in broadcast channels to various receiver types with Multimedia capabilities.

Various possibilities for transmitting information are incorporated into a common transport mechanism for different DAB data channels, so that the access to Multimedia content is unified within the DAB system.

MOT ensures interoperability between:

- different data services and user application types;
- different receiver device types and targets;
- equipment from different manufacturers.

Each data service has an associated user application specification, and that specification includes the transport mechanisms for the data content. If the user application uses files of information then these are best transported using the MOT protocol layered onto the DAB transport mechanisms for packet mode and X-PAD.

The MOT protocol allows objects of a finite length from an information source, i.e. the content/service provider to be conveyed to a destination, i.e. the terminal, as shown in figure 1, where in terms of MOT:

The Content/Service provider is capable of processing various types of Multimedia content (e.g. picture and text files) in an appropriate way, so that this data is compliant with the MOT specification and can be fed into a DAB Ensemble multiplexer;

The Terminal is fed from an MOT decoder capable of processing the multimedia content of a DAB Ensemble in an appropriate way, so that it is:

- decoded and presented to the user; or
- forwarded to a following entity, which then processes the content.

The definition of interfaces between the different entities is not within the scope of the MOT specification.

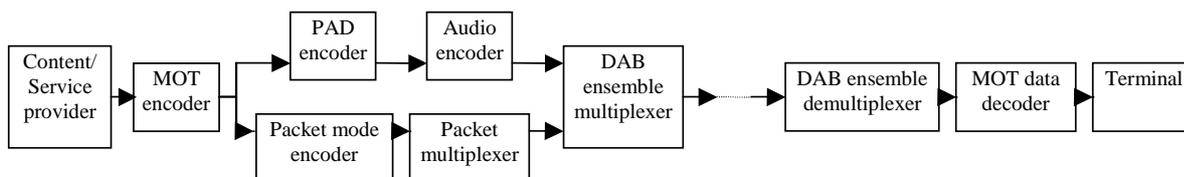


Figure 1: Overview of MOT encoding and decoding

MOT interconnects the closed and well-defined world of DAB to the open world of Multimedia services with its large variety of systems and data formats. It comprises functionality to carry information to the terminal, and ultimately to the user.

In addition to the Multimedia transport the MOT protocol also supports handling of the Multimedia objects (e.g. object identification or object management on receiver side) and provides additional information that can support a user application.

MOT does not cover issues specific to runtime environments to control Multimedia services, i.e. the interpretation and execution of object code, pseudo code or script languages. This shall be included in the particular user application.

The structure of the Multimedia content is user application specific and not subject to standardization within the present document.

4.3 Receiver architecture reference model

An example decoding process for MOT objects is shown in figure 2 (data flow top-down).

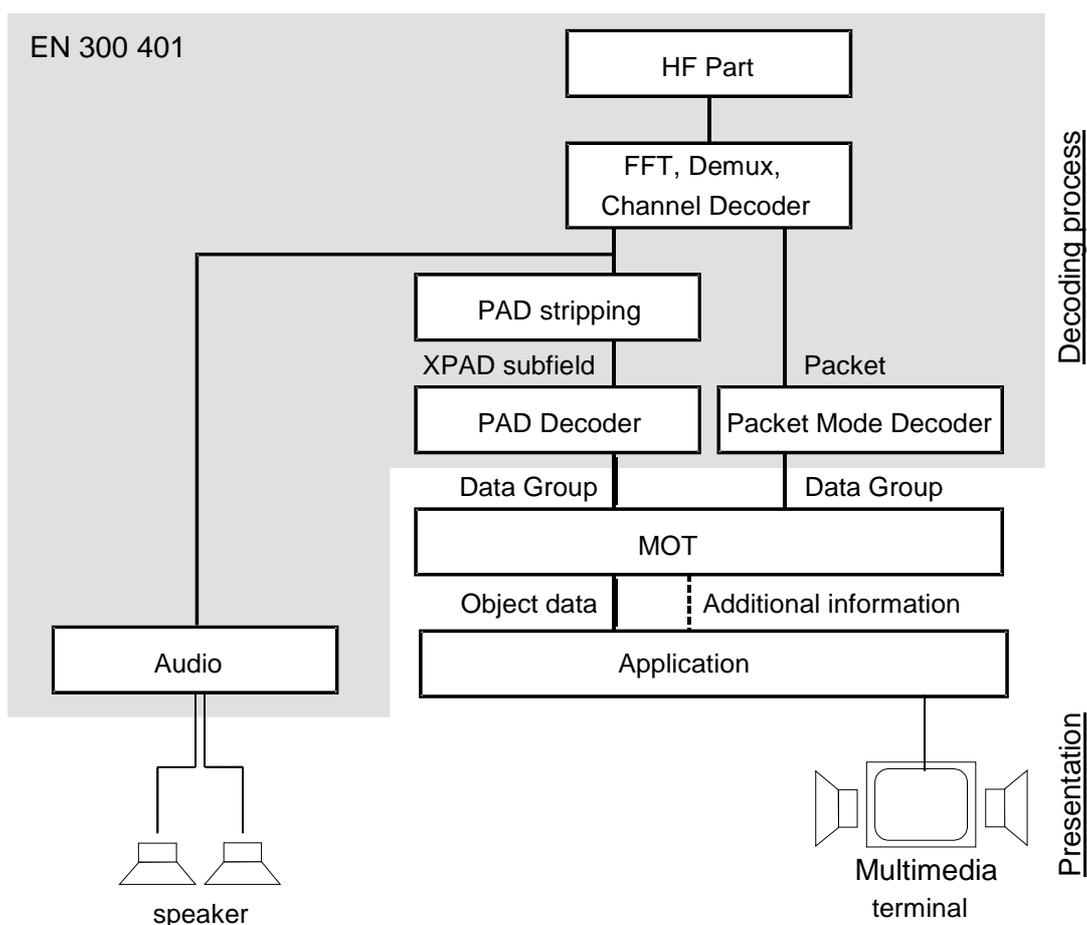


Figure 2: Example scheme for the data decoding part of a DAB receiver

Parts within the grey background (HF part, FFT/demux/channel decoder, PAD stripping, PAD decoder, packet mode decoder and audio decoder) are defined in EN 300 401 [1]).

Interface to the MOT decoder: Communication between PAD/packet mode decoder and MOT decoder uses complete MSC data groups (see EN 300 401 [1]). The session header of a data group cannot be omitted, although it is optional in the DAB specification, since it carries the *TransportId*, which is necessary to reassemble the MOT objects.

Additional information: Additional information is carried in the MOT header information. It is decoded by the MOT decoder and forwarded to the user application decoder.

Object data: Object data is carried in the MOT body.

5 Structural description

This clause describes the different operations needed in order to transmit a file/object or a set of files/objects using the MOT protocol.

MOT provides two modes of operation - MOT directory mode and MOT header mode. The MOT mode determines how management data is carried.

- The MOT directory mode permits the content provider to manage a set of files/objects on receiver side (file/object addition, deletion and modification). In MOT directory mode all management data from all broadcast files/objects is combined in one single MOT entity, the "MOT directory". The MOT directory is sent "in parallel" to the files/objects it describes.
- MOT header mode can be used for user applications that use one single file/object at a time. In MOT header mode the management data of the one and only MOT object is contained in the MOT entity "MOT header". The MOT header is sent in advance of the file/object it describes.

The user application specification defines the MOT transport mode that shall be used to convey the user application data.

The first step in the transmission process is to identify the file and to create the header information. The header information contains both pure file identification and additional information. The file is referred to as the MOT body. At this stage the header information and body correspond to an MOT object.

The header information is separated from the body during transportation in order to:

- have the possibility to repeat the header information several times before and during the transmission of the body (which is useful when transmitting long objects);
- send the header information in advance in order to give the receiver the opportunity to "be prepared in advance" to the data that is going to be received;
- send the header information unscrambled when the body is scrambled.

The data flow at the transmitter side is shown in figure 3. The different data files that should be transferred via DAB are first processed in the MOT encoder, producing MOT objects. Then the PAD or packet mode specific coding is applied. For all the subsequent stages see EN 300 401 [1]. A packet mode sub-channel may contain a number of service components (some of them MOT based), separated by the packet address. An X-PAD channel may carry a number of user applications (some of them MOT based) in parallel, separated by the X-PAD application type values. Finally the sub-channels (stream mode audio, stream mode data, packet mode) are multiplexed into the DAB ensemble.

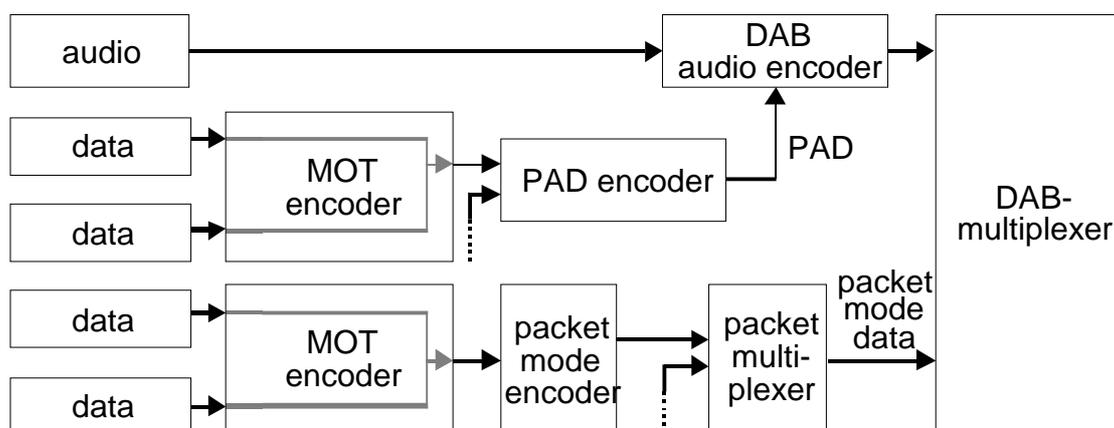


Figure 3: Data transfer in DAB using MOT - data flow

Figure 3 can be converted into a layered scheme indicating the steps, which have to be performed (see figure 4).

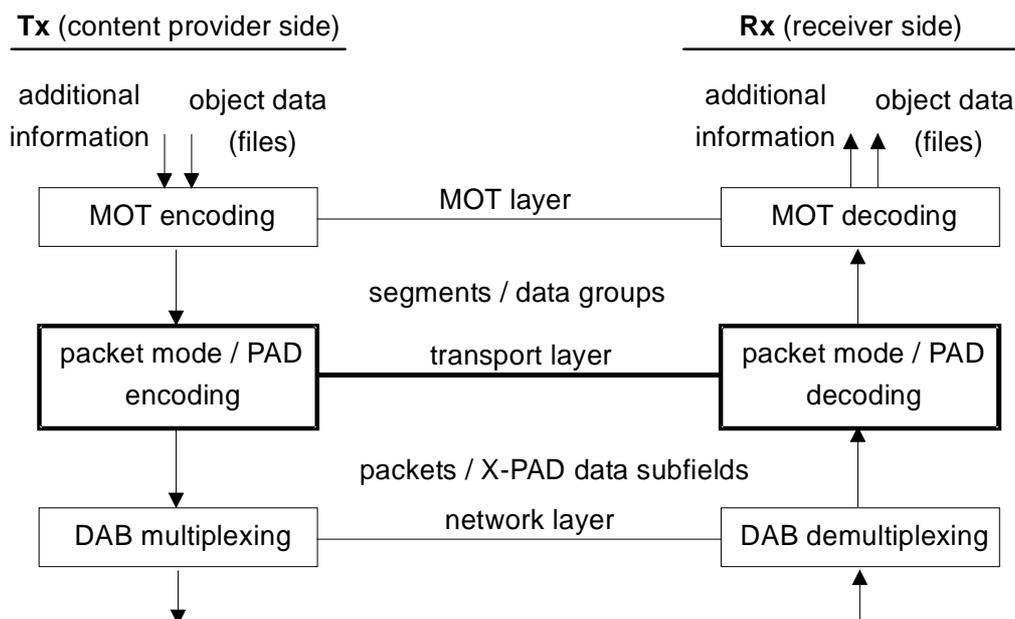


Figure 4: Data transfer in DAB using MOT - protocol layers

The coding procedure starts at the object level, which stands for the files (including some management information) to be transferred and processed further.

MOT encoding generates the complete MOT objects including the additional header information and transforms these entities (MOT body, MOT directory or MOT header) into segments of an appropriate size for the lower protocol layers.

Packet mode/PAD encoding transforms these segments into MSC data groups and further into packets, which fit into the container provided by DAB (X-PAD data subfields, packet mode packets).

DAB encoding and multiplexing handles the output of the PAD/packet mode encoder and supplies either a complete packet mode sub-channel or fills the X-PAD fields of the audio stream.

5.1 Segmentation of MOT entities

The lowest common structure for the two different DAB transport mechanisms for which MOT is defined (Packet Mode and X-PAD) is MSC data groups. This data group structure is also mapped to X-PAD transport for compatibility reasons. It is therefore the goal of the segmentation on MOT level to map the MOT entities into MSC data groups.

The header information and body are transported in different MOT entities and therefore the segmentation will apply independently on header information and body. MOT entities will be split up in segments with equal size. Only the last segment may have a smaller size (to carry the remaining bytes of the MOT entity). Every MOT entity (e.g. every MOT body) can use a different segmentation size.

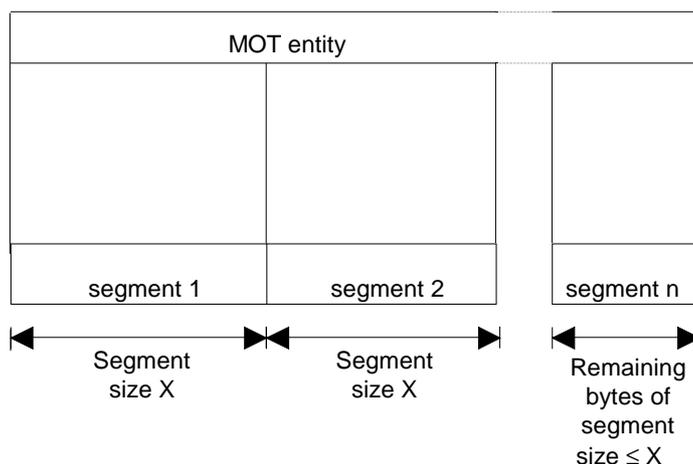


Figure 5: Segmentation of MOT entities

To elaborate a proper segmentation strategy the following considerations should be taken into account:

- minimize the overhead;
- improve the robustness of the transmission;
- facilitate the segment management on MOT decoder side.

The Segmentation header (see clause 5.1.1) is attached to all segments and then every segment of the above-mentioned MOT entities (MOT body, MOT directory, MOT header) is mapped to an MSC data group with the appropriate data group type (see clauses 5.1.2 to 5.1.4) for later transport in one or more packets or X-PAD data subfields.

Every MOT decoder shall ignore MSC data groups with data group types not supported by this MOT decoder. This behaviour is required to permit future extensions (that might use additional data group types) of the MOT protocol.

A data group shall contain a data group header, a Session header, a data group data field and an optional data group CRC. The structure of a data group is shown in EN 300 401 [1].

The user access field in the Session header (see EN 300 401 [1]) is not optional if MOT segments are carried in MSC data groups. It cannot be omitted, since this field contains the *TransportId*, necessary for MOT object transfer. The use of the MSC data group CRC is strongly recommended.

The link between header information in an MOT header (data group type 3) or MOT directory (data group type 6 or 7) and its MOT body (data groups type 4 or 5) is established by the *TransportId*.

In MOT header mode the *TransportId* of the MOT body is the same as the *TransportId* of its MOT header.

In MOT directory mode the *TransportId* of the MOT body is attached to the header information for this MOT object inside the MOT directory. If scrambled MOT objects are used, the CA messages related to this object also have the same *TransportId*.

As long as the *TransportId* of an MOT entity is not changed, the segmentation size of this MOT entity shall remain the same.

Figure 6 shows all necessary steps to split MOT entities and map them to MSC data groups, see clause 5.3.3 in EN 300 401 [1].

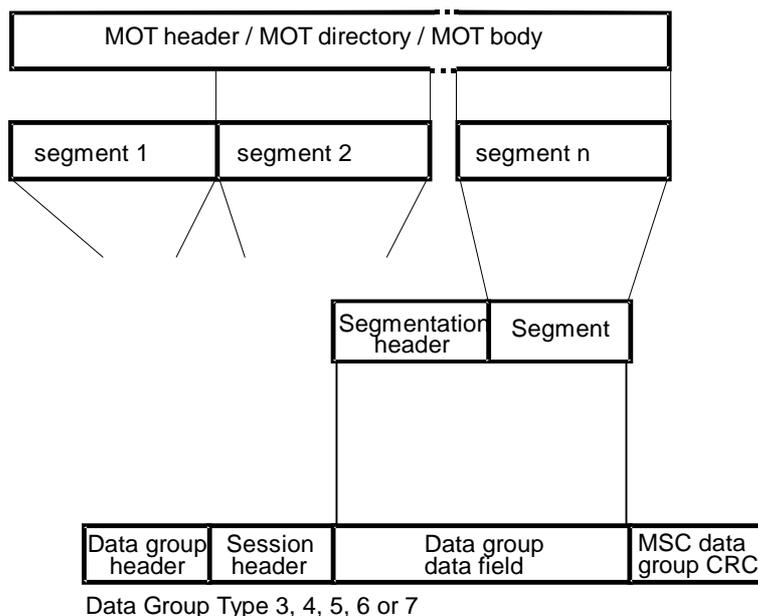


Figure 6: Segmentation of MOT header, MOT directory or MOT body

5.1.1 Segmentation header

The **Segmentation header** (see figure 7) shall be attached to each segment of an MOT entity and contains information about the size of the segment and the remaining repetitions of the entity.

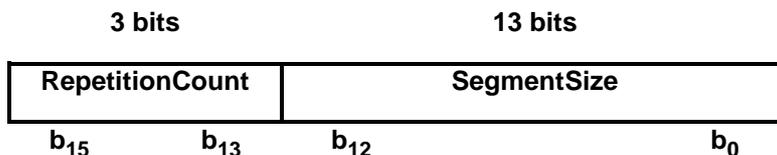


Figure 7: Segmentation header

RepetitionCount: This 3-bit field indicates, as an unsigned binary number, the remaining transmission repetitions for the current entity - repetition on object level (if in MOT header mode, see figure 12) or repetition on entity level (if in MOT directory mode, see figure 18). Exceptionally, the code "111" shall be used to signal that the repetition continues for an undefined period (> 6 times).

SegmentSize: This 13-bit field, coded as an unsigned binary number, indicates the size of the segment data field in bytes. The maximum length which can be signalled is 8 189 bytes according to the limited total length of the MSC data group data field (8 191 bytes), so that both, the Segment and the Segmentation header fit into one MSC data group.

Note that some MOT decoders may ignore both parameters of the Segmentation header.

5.1.2 Segmentation of the MOT body

If conditional access on MOT level is applied, then scrambled MOT body segments shall be transported in MSC data group type 5. In all other cases (no scrambling on MOT level or unscrambled MOT body segments) the segments of the MOT body shall be transported in MSC data group type 4.

Segmentation is applied considering the size of the MOT body and the segmentation strategy.

5.1.3 Segmentation of the MOT directory

The segments of an uncompressed MOT directory shall be transported in MSC Data Group type 6. The segments of a compressed MOT directory shall be transported in MSC Data Group type 7.

Segmentation is applied considering the size of the MOT directory and the segmentation strategy.

The MOT directory is not scrambled if conditional access on MOT level is applied.

The link between a directory entry inside an MOT directory (data group type 6 or 7) and the MOT body is established by the *TransportId*. The *TransportId* attached to the header information and its MOT body is the same. If scrambled MOT objects are used, the CA messages related to this object also have the same *TransportId*.

5.1.4 Segmentation of the MOT header

The segments of the MOT header shall be transported in MSC Data group type 3.

In order to enable easier access to the header information and to reduce the memory demand in the reassembly unit of the MOT decoder, it is recommended to send the MOT header in one MSC data group (which is equivalent to no segmentation).

Conditional access on MOT level is not possible for MOT header mode; therefore the MOT header is never scrambled on MOT level.

5.2 Transporting MOT segments - network level

The coding of the data on network level is described in detail in EN 300 401 [1], therefore clauses 5.2.1 and 5.2.2 only explain the actual mapping of the MSC data groups obtained on the transport layer into the packet mode packets or X-PAD data subfields.

5.2.1 Packet mode

The MSC data groups containing MOT data are transmitted in one or more packets sharing the same packet address (see EN 300 401 [1]).

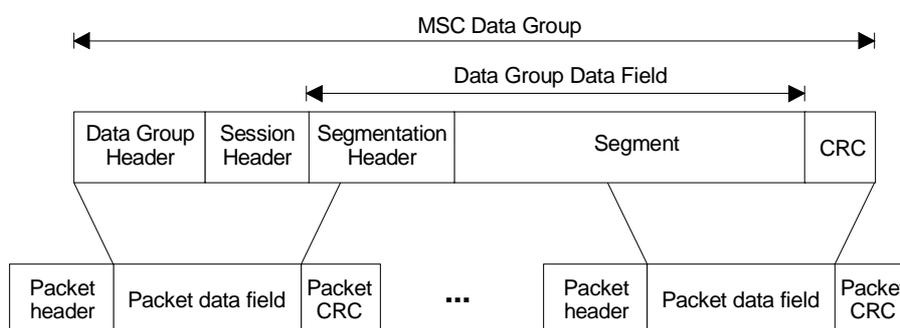


Figure 8: Relationship between a MSC data group and a sequence of packets

The command flag on packet level is used to identify conditional access data. The command flag is 0 for MSC data group types 3, 4, 6 and 7 (MOT header, unscrambled MOT body, uncompressed MOT directory and compressed MOT directory). The command flag is 1 for conditional access data carried with MSC data group type 1 and 5 (ECM/EMM data and scrambled MOT body for conditional access on MOT level).

5.2.2 X-PAD

The MSC data groups containing MOT data are transmitted in one or more X-PAD data subfields (see EN 300 401 [1]) using MSC data group structure.

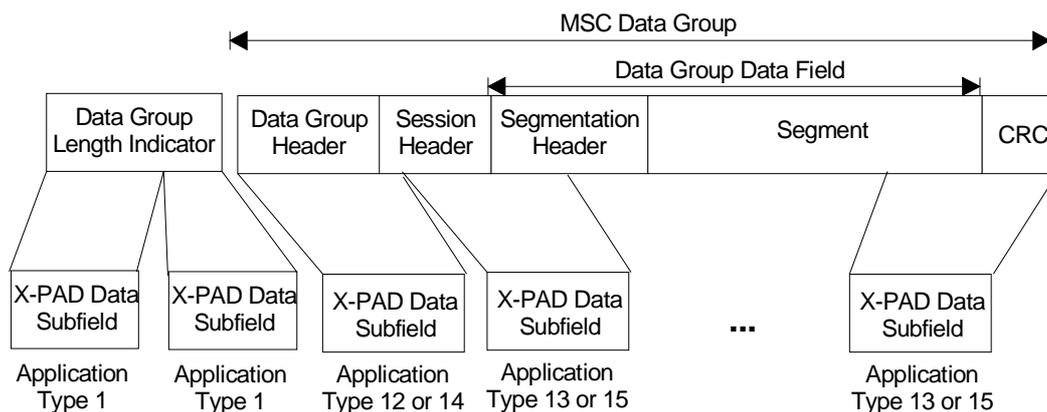


Figure 9a: Example for transportation of MSC data groups in X-PAD data subfields in case of short X-PAD

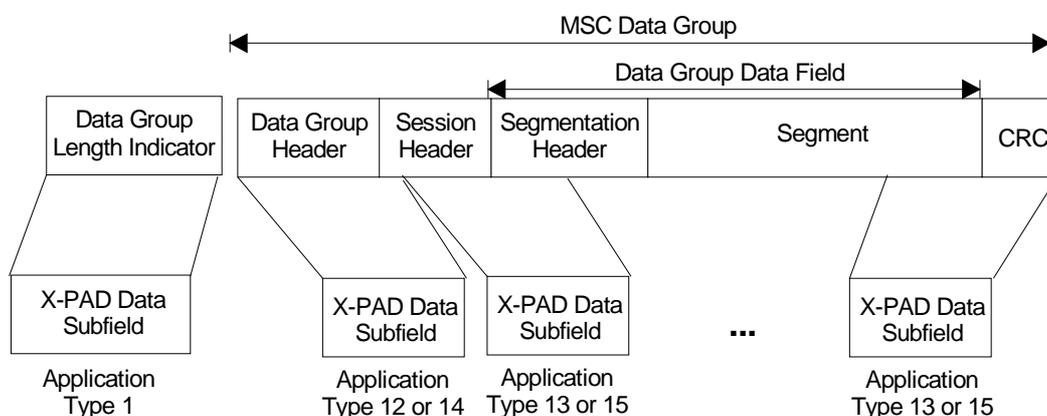


Figure 9b: Example for transportation of MSC data groups in X-PAD data subfields in case of variable size X-PAD

A complete specification of the transport signalling in X-PAD is given in EN 300 401 [1].

MOT transport in X-PAD uses at least two X-PAD application types (in addition to the MSC data group length indicator with X-PAD application type 1).

If conditional access on MOT level is applied then a second pair of X-PAD application types is used. Similar to the command flag in packet mode the use of two pairs of X-PAD application types is used to identify conditional access data. The first pair of X-PAD application types is used for MSC data group types 3, 4, 6 and 7 (MOT header, unscrambled MOT body, uncompressed MOT directory and compressed MOT directory). The second pair of X-PAD application types is used for conditional access data carried with MSC data group type 1 and 5 (ECM/EMM data and scrambled MOT body).

In the former MOT specification the MOT protocol was limited to one single MOT based user application in X-PAD. MOT in X-PAD was restricted to X-PAD application types 12/13 and 14/15 (conditional access data). According to the current MOT specification, multiple MOT based user applications can be carried in X-PAD (using additional X-PAD application type values).

MOT decoders limited to the former MOT specification can only decode user applications in X-PAD that use X-PAD application types 12/13 or 14/15. For backwards compatibility it is recommended to use X-PAD application types 12/13 (and 14/15 if conditional access on MOT level is applied) for the primary MOT user application carried in X-PAD.

5.3 Transmission Mechanisms

When transmitting data in a radio broadcast system, the content provider always has to take in consideration that the terminal may miss data due to:

- The receiver being switched off, out of signal or tuned to another service or ensemble.
- Bad reception due to the conditions of the radio channel (bit errors).

Without an interaction channel the receiver does not have the possibility to actively demand repetition of lost data. Therefore MOT uses some mechanisms, which permit data to be repeated and so allow the receiver several opportunities to receive objects and to improve the reception probability:

- Repetition on data group (MOT segment) level.
- Repetition on MOT object/entity level.

Repetition means that a data group or MOT object/entity is broadcast again, with exactly the same content. The term Retransmission means that an MOT object (identified by its ContentName) is broadcast again, but possibly with an updated content.

Reception errors may cause the receiver to fail in decoding the data, therefore it is strongly recommended to use one or more of the mechanisms explained below to improve the reception probability, although the use of these mechanisms decreases the useful bit rate.

The following clauses detail the main differences and characteristics of the single object transmission (MOT header mode) and multiple object transmission (MOT directory mode) mechanisms. For further details regarding MOT header mode and MOT directory mode see clause 7.

5.3.1 Single object transmission (MOT header mode)

The single transmission scheme (see figure 10) is useful when only one object is valid at a time, for example in a slide show.



Figure 10: Single object transmission

5.3.1.1 Repetition on object level

An object can be transmitted several times so that the receiver can replace MOT segments of an object, lost due to reception errors, with the repetition of the same MOT segments if they are received without reception errors. Repetition on MOT object level is signalled in the MOT segmentation header (see clause 5.1.1). Figure 11 shows the repetition method based on transmitting the entire object a number of times.

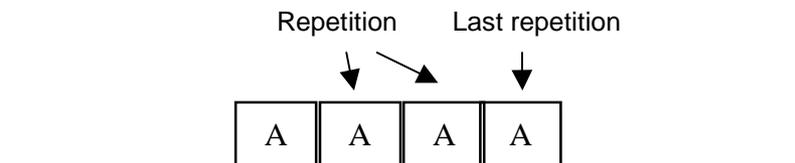


Figure 11: Single object transmission with repetitions

Figure 12 explains how the segments of the object (consisting of MOT header ("H") and MOT body ("Body")) are repeated.

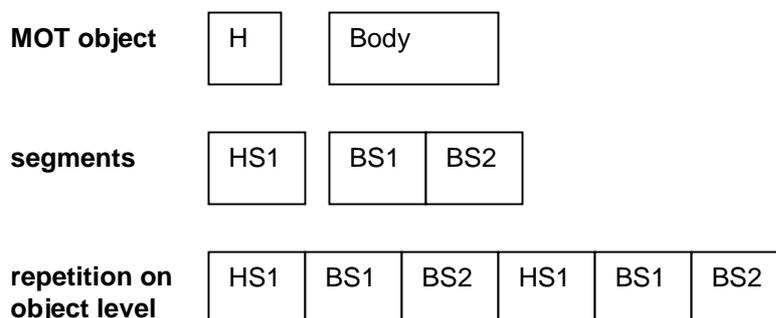


Figure 12: Repetition on object level (example)

5.3.1.2 Repetition of MSC data groups/MOT segments

Segments of an object can be transmitted several times so that the receiver can replace those segments, lost due to reception errors, with the repetition of the same object segments received without errors. Repetition on MSC data group level is signalled in the MSC data group header (see EN 300 401 [1]). Figure 13 shows the repetition method based on transmitting every segment of an object (consisting of MOT header ("H") and MOT body ("Body") segments) a number of times.

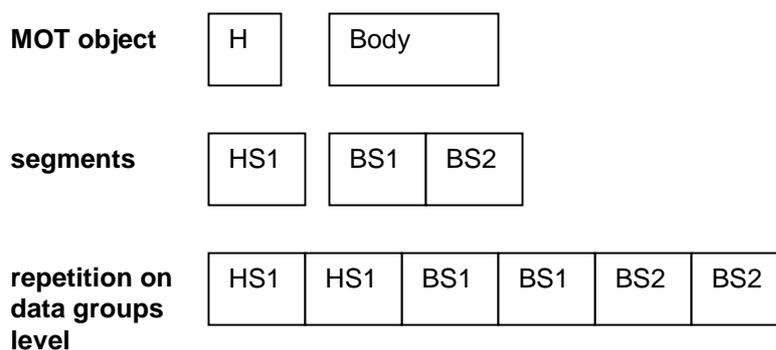


Figure 13: Repetition of MSC data groups/MOT segments (example)

5.3.1.3 Repeated transmission of header information (combined with repetition on object level)

During the transmission of body segments (MSC data groups type 4 or 5) of large objects it can be useful to transmit the complete header or part of the header information carried in MSC data groups type 3 (see figure 14) multiple times. This allows the data decoder to detect the object even if it has not received the start of the object transmission. Provided that the object is repeated the data decoder can then complete the missing segments during the next repetition of the object.

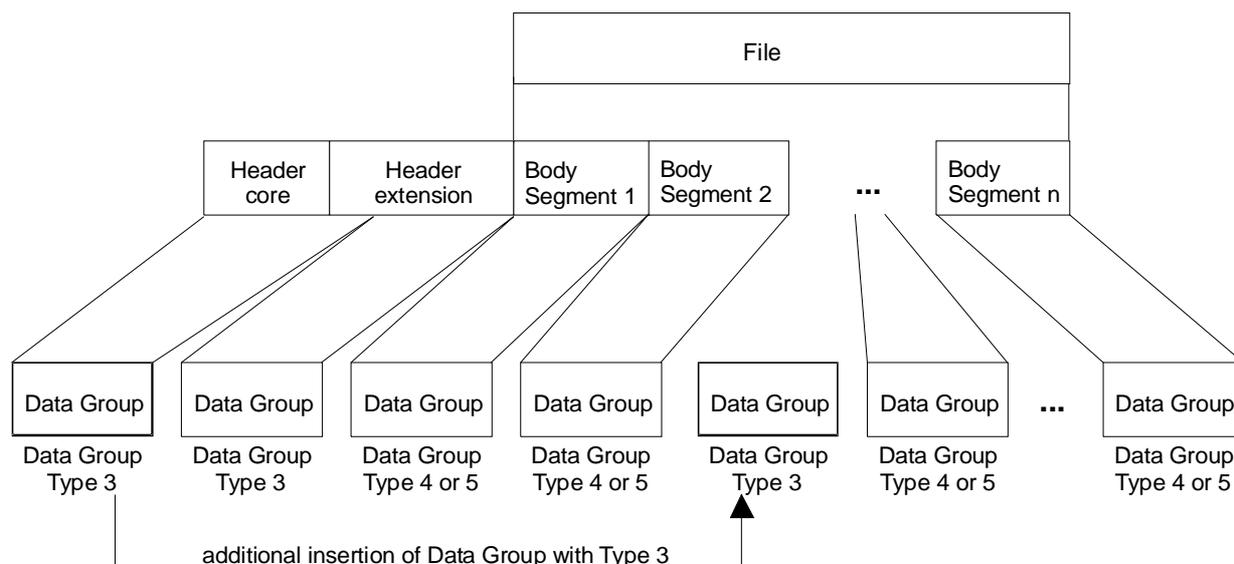


Figure 14: Repeated transmission of header information

5.3.2 Multiple object transmissions (MOT directory mode)

Multiple object transmission (data carousel functionality) is intended for user applications that need to have several objects available on the terminal at the same time. An example for such a user application is a broadcast web site.

Each MOT body is transmitted several times in a cyclic manner with a cycle time between each transmission (see figure 15). The cycle time between subsequent transmissions of an MOT body may vary.

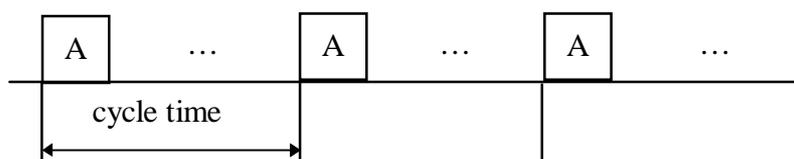


Figure 15: Multiple object transmission

Within a cycle (see figure 16), repetition on MSC data group (MOT segment) level (see figure 17) or on MOT entity level (see figure 18) can be used to ensure the reception. Repetition on MSC data group level is signalled in the MSC data group header (see EN 300 401 [1]); repetition on MOT entity level is signalled in the MOT segmentation header (see clause 5.1.1). In a new cycle the object is retransmitted. The content of the object can be the same or can be updated.

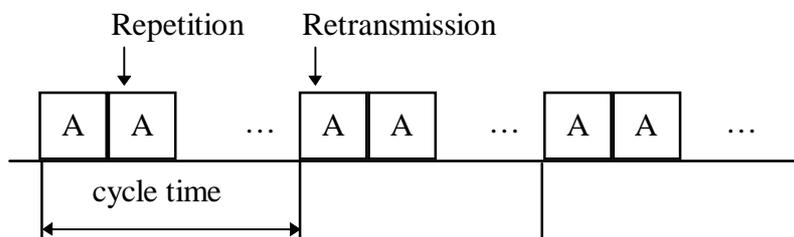


Figure 16: Multiple object transmission with repetitions

Figures 17 and 18 explain how MOT entities (MOT directory or MOT bodies) can be repeated.

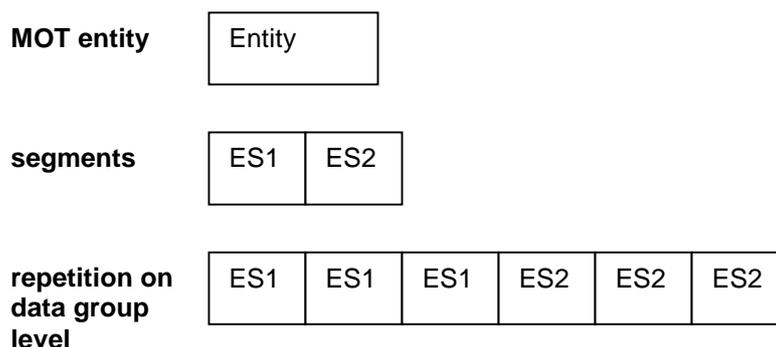


Figure 17: Entity repetition on MSC data group level (example)

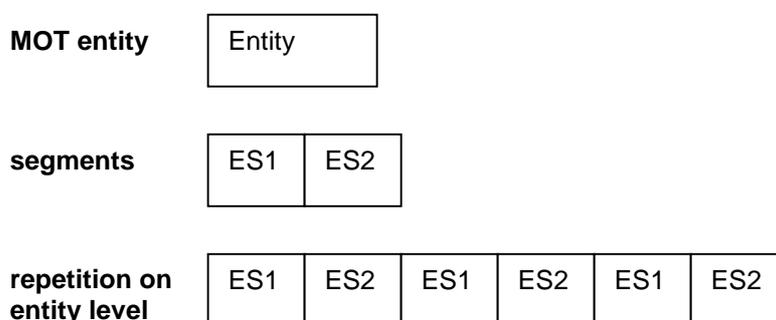


Figure 18: MOT entity repetition on entity level (example)

The different objects in the cyclic MOT stream are identified by their *ContentNames*. If in a retransmission the header information, the body content or the *SegmentSize* of an object have been changed, the *TransportId* shall be changed. If nothing is changed, the *TransportId* should remain the same.

It is recommended to transmit the most important MOT bodies (e.g. the HTML pages on the top of the hierarchy or the most likely visited pages of a broadcast web site) more frequently than the others to improve the access to the service. To broadcast some objects more frequently than others interleaving of MOT entities is used (see clause 5.3.2.1).

5.3.2.1 Interleaving MOT entities in one MOT stream

Interleaving permits the transfer of data groups of different MOT bodies and/or the MOT directory in parallel.

With the MOT protocol it is possible to transmit several MOT entities (i.e. MOT directory and MOT bodies) in parallel in one single data channel (i.e. in one MOT stream). The different MOT entities are separated by their data group type and their *TransportId* (see EN 300 401 [1]). It is very common to interleave multiple MOT bodies and in addition interleave them with the MOT directory.

Interleaving can for instance be used to insert high priority objects (e.g. the MOT directory or an entry page of a broadcast web site) into the MOT stream during the transmission of big objects with a long transmission time.

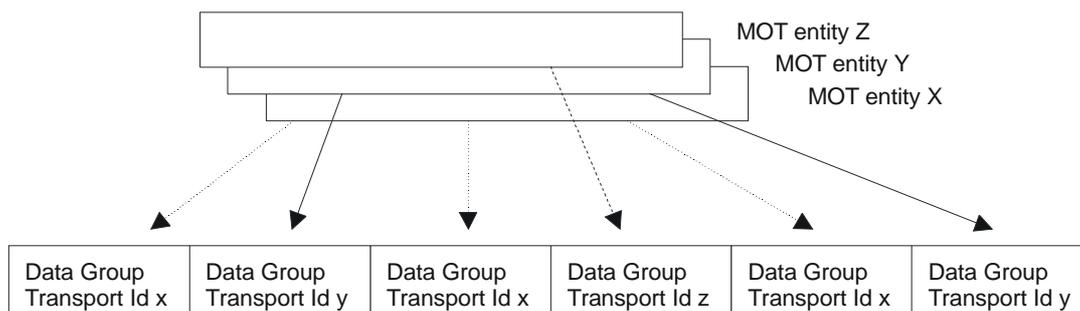


Figure 19: Interleaving of MOT entities on MSC data group level

6 MOT header information

The MOT header information consists of two parts: the header core and the header extension. The header core is a set of four parameter fields that shall be specified, while the header extension is a variable length field that may contain an arbitrary number of "parameters" to be associated with each object.

This clause applies to both the MOT header mode and the MOT directory mode. It defines the general structure and format of MOT parameters that describe an MOT object.

6.1 Header core

The header core contains information about the size and the content type of the object, so that the receiver can determine whether it has system resources to decode and present the object or not.

The header core fields are `BodySize`, `HeaderSize`, `ContentType` and `ContentSubType`. The first two fields indicate the length, in bytes, of the body and header information respectively. The `ContentType` and `ContentSubType` pair of fields is used to indicate the object type (i.e. the content type of the body), where that type identifier is taken from an enumerated list.

The header core shall be coded as shown in figure 20.

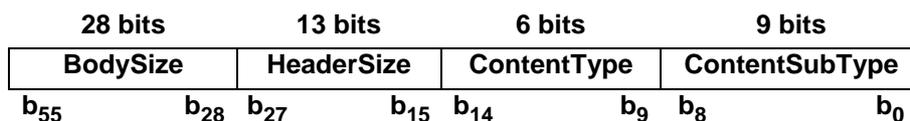


Figure 20: Structure of the header core

BodySize: This 28-bit field, coded as an unsigned binary number, indicates the total size of the body in bytes.

If the body size signalled by this parameter does not correspond to the size of the reassembled MOT body, then the MOT body shall be discarded.

HeaderSize: This 13-bit field, coded as an unsigned binary number, indicates the total size of the header information in bytes including the header core size of 7 bytes.

ContentType: This 6-bit field indicates the main category of the body's content. The interpretation of this field shall be defined in TS 101 756 [7], table 17.

ContentSubType: This 9-bit field indicates the exact type of the body's content depending on the value of the field `ContentType`. The interpretation of this field shall be defined in TS 101 756 [7], table 17.

In many user applications, the list of `ContentType` and `ContentSubType` may be sufficient to define all the possible types of objects that may be used by the user application. However, since the enumerated list is necessarily constrained to the set of types that have been registered, a user application may choose to use an alternative mechanism to determine the object type, if needed, such as using the MOT parameter `MimeType` (see clause 6.2.2.1.2).

Even if a user application does not use `ContentType` and `ContentSubType` to indicate the type of an object (e.g. because the user application uses the MOT parameter `MimeType`; see below), the fields `ContentType` and `ContentSubType` shall correctly indicate the body's content type. If a corresponding value of the fields `ContentType` and `ContentSubType` is not available in the enumerated list defined within TS 101 756 [7], table 17, then both fields shall be set to 0 to indicate "general data/object transfer".

For the `ContentType` "application" the body's content type identified by `ContentType/ContentSubType` depends on the user application. It is possible that two user applications use the same value for `ContentSubType` but assign a totally different meaning to it. If the `ContentType` "application" is signalled, then the decoding of `ContentType/ContentSubType` always has to take into account which user application provides the data.

6.2 Header extension

The header extension provides information that supports the handling and transport of the objects (e.g. object identification or object management) and provides additional information for the user application decoder.

The header extension parameters describe several attributes of the object. Some of these parameters may occur more than once as described separately for the different parameters.

The user application defines which MOT parameters are used when transporting this user application's data. The user application may also restrict the permitted values for MOT parameters (e.g. the parameter `CompressionType` might permit multiple compressed data formats, but the user application might choose to permit just one single way to compress data).

Note that in MOT directory mode the MOT directory contains all header information of all MOT objects within the data carousel. The size of the MOT directory has a strong impact on its cycle time and the content provider should try to keep the MOT directory as small as possible. Therefore the content provider should try to keep the header information of his MOT objects as small as possible (i.e. by using short `ContentNames`).

The general structure of the header extension is shown in figures 21 and 22.



Figure 21: General structure of the header extension

Each parameter in the header extension consists of a length indicator, a parameter identifier and a data field. The parameter identifier determines how the data field is to be interpreted.

Every parameter of the MOT header extension shall be coded as shown in figure 22.

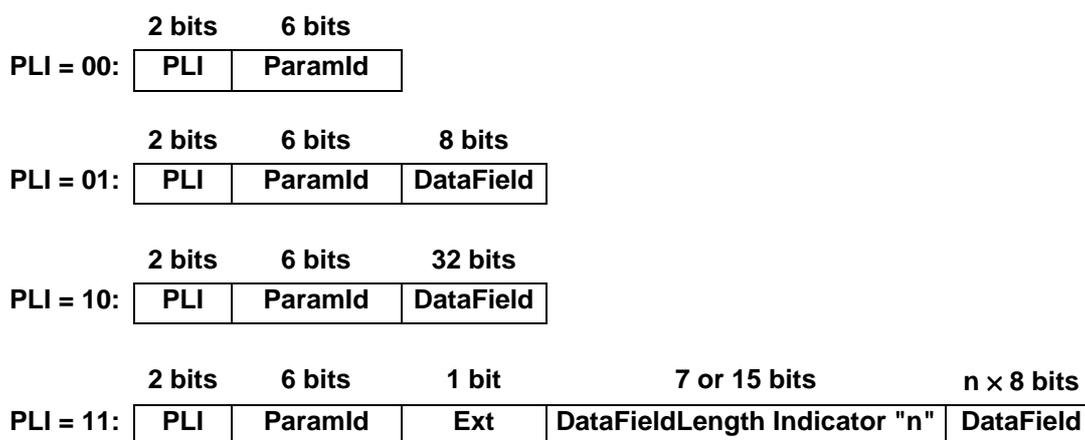


Figure 22: Structure of the header extension parameter

PLI (Parameter Length Indicator): This 2-bit field describes the total length of the associated parameter. The following definitions apply:

- 0 0 total parameter length = 1 byte, no DataField available;
- 0 1 total parameter length = 2 bytes, length of DataField is 1 byte;
- 1 0 total parameter length = 5 bytes; length of DataField is 4 bytes;
- 1 1 total parameter length depends on the DataFieldLength indicator (the maximum parameter length is 32 770 bytes).

ParamId (Parameter Identifier): This 6-bit field identifies the parameter. The coding is defined in table 2 at the end of clause 6.

Ext (ExtensionFlag): This 1-bit field specifies the length of the DataFieldLength Indicator and is coded as follows:

- 0: the total parameter length is derived from the next 7 bits;
- 1: the total parameter length is derived from the next 15 bits.

The ExtensionFlag is only present if the PLI field is set to "11".

DataFieldLength Indicator: This field specifies as an unsigned binary number the length of the parameter's DataField in bytes. The length of this field is either 7 bits or 15 bits, depending on the setting of the ExtensionFlag.

The DataFieldLength Indicator is only present if the PLI field is set to "11".

DataField: This field contains the parameter data and is only present if the contents of the PLI field is either 01, 10 or 11.

The PLI field can be used to efficiently code some commonly used DataField lengths (0 bytes, 1 byte or 4 bytes). A DataField length of 0 bytes can be encoded with the PLI field set to "00" or with the PLI field set to "11" and a DataFieldLength Indicator set to 0 (bytes). A DataField length of 1 byte can be encoded with the PLI field set to "01" or with the PLI field set to "11" and a DataFieldLength Indicator set to 1 (byte). A DataField length of 4 bytes can be encoded with the PLI field set to "10" (4 bytes) or with the PLI field set to "11" and a DataFieldLength Indicator set to 4 (bytes). The PLI field (possibly together with DataFieldLength indicator field) shall only be used to determine the length of the DataField; the interpretation of the DataField shall not depend on the value of the PLI field!

6.2.1 Future expansion of the parameter data field

The length of a parameter's DataField is described by the Parameter Length Indicator (PLI) and the DataFieldLength Indicator. The generic structure and flexibility of MOT allows future expansions of the parameter data field.

Therefore each parameter with a fixed DataField length in one revision of the MOT standard can be expanded in a later revision of the MOT standard by appending new information at the end of the data field (without changing the definition of the original data field; see figure 23). A decoder working according to the older revision of the MOT standard will still be able to extract all the expected data, while a decoder working according to the later revision can additionally interpret the extended information.

2 bits	6 bits	1 bit	7 bits	8 bits	$N \times 8$ bits
PLI	ParamId	Ext.	Length	CompressionType	New parameter field
11	000110	0	1 + N		

Figure 23: Example for the expansion of a defined parameter

An MOT decoder evaluating an MOT parameter with fixed DataField length shall not check if the MOT parameter length is equal to what the MOT decoder expects. It shall always check if the length of the DataField is equal *or greater* than what the MOT decoder expects!

6.2.2 Parameters of the header extension for MOT header mode and MOT directory mode

The header extension contains transport protocol specific and user application specific parameters. The following clauses describe the transport specific parameters grouped by functionality.

The user application defines which MOT parameters are used and the user application definition might also restrict the range of values permitted for MOT parameters.

6.2.2.1 MOT Basic transport parameters

The MOT basic transport parameters are used to identify and describe the MOT body.

6.2.2.1.1 ContentName

The parameter `ContentName` is used to uniquely identify an object. At any time only one object with a certain `ContentName` shall be broadcast.

The `DataField` of this parameter starts with a one byte field, comprising a 4-bit character set indicator (see TS 101 756 [7], table 19 for the list of permitted character sets) and a 4-bit `Rfa` field. The preferred character set is ISO latin1. The following character field contains a unique name or identifier for the object. The total number of bytes is determined by the `DataFieldLength` indicator minus one byte.

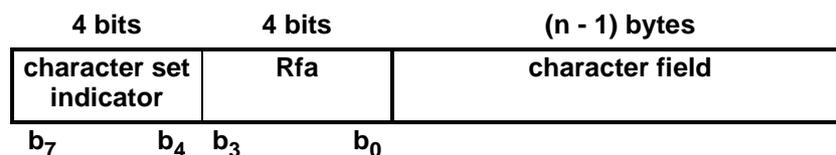


Figure 24: Coding of the MOT parameter ContentName

Hierarchical structures shall use a slash "/" to separate different levels. No system specific restrictions shall be applied. Forward slashes ("/") inside the `ContentName` separate levels and slashes are not permitted for any other meaning than this.

Unless explicitly required by the user application definition, the `ContentName` does not have to be a valid filename for any operating system. Thus a general purpose MOT decoder can not use the `ContentName` as (part of) the filename where it stores the MOT body. Unique filenames (e.g. containing the *TransportId*) should be used. The MOT decoder must be able to map the `ContentNames` required by the user application to the internally used filename or data structure that holds the MOT body.

6.2.2.1.2 MimeType

In HTTP, the type of an object is indicated using the Multi-purpose Internet Mail Extensions (MIME) [4] mechanism. MIME strings categorize object types according to first a general type followed by a specific format, e.g. "text/html", "image/jpeg" and "application/octet-stream".

NOTE: The basic MIME string may optionally be followed by a ";" and a parameter list. This mechanism is typically used to indicate character sets for text types.

In order to correctly present MOT objects, it is required for the receiver to be able to determine the type of the object. Some content types may be signalled using the `ContentType` and `ContentSubType` fields of the MOT header information. However, this mechanism is unsuitable for supporting as yet unspecified MIME types, and so limits the range of types that may be supported *even when the receiver is PC based*. In order to overcome this limitation, the `MimeType` parameter may be used to supply a MIME type string for each object.

The user application definition indicates how the user application decoder is informed about the type of an MOT object. The content type of an MOT object can be indicated using the `ContentType/ContentSubType` field of the MOT header core, this MOT parameter `MimeType` or by other means such as the "file name extension".

If a user application definition requires that the MOT parameter `MimeType` is used to indicate the type of the MOT object, then no user application decoder shall use the "file name extension" to derive the type of an MOT object!

If a user application uses the MOT parameter `MimeType` to indicate the type of an MOT object, then the fields `ContentType` and `ContentSubType` of the MOT header core (see clause 6.1) shall correctly indicate the body's content type. If a corresponding value of the fields `ContentType` and `ContentSubType` is not available in the enumerated list defined within TS 101 756 [7], table 17, then both fields shall be set to 0 to indicate "general data/object transfer".

The `DataField` of this parameter carries the MIME type string appropriate to the object, see [4].

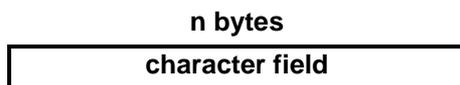


Figure 25: Coding of the MOT parameter `MimeType`

6.2.2.1.3 CompressionType

The `CompressionType` parameter is used to indicate that an object has been compressed and which compression algorithm has been applied to the data. The `DataField` of this parameter carries a one byte identifier (`CompressionId`) for the compressed data format. If new compressed data formats are to be used, a new `CompressionId` shall be obtained from and registered with the WorldDAB Information and Registration Centre.

The registered compressed data formats shall be defined in TS 101 756 [7], table 18.

Even an MOT decoder that does not support any compression shall check this parameter to determine if an MOT body is compressed.

6.2.3 Parameters of the header extension for MOT directory mode only

6.2.3.1 MOT caching support parameters

These parameters can be used to support caching of MOT objects on receiver side. The caching functionality is described in detail in clause 8.1.

6.2.3.1.1 Expiration

The parameter `Expiration` indicates how long an object can still be used by the MOT decoder after reception loss. The size of the `DataField` determines if an absolute or a relative expire time is specified.

The MOT directory extension parameter `DefaultExpiration` described in clause 7.2.4.3 (if present) defines a default value for all MOT objects that do not provide the MOT header information parameter `Expiration`. The default value defined by `DefaultExpiration` only applies to an MOT object if no MOT header information parameter `Expiration` is provided for this MOT object.

If the MOT header information parameter `Expiration` is provided for an MOT object, it defines the expire time of the object. If `Expiration` is not provided for the object, then the MOT directory extension parameter `DefaultExpiration` defines the expire time of the object. If neither the MOT header information parameter `Expiration` is provided for an MOT object nor the MOT directory extension parameter `DefaultExpiration` is provided as a default for all objects, then the MOT object never expires.

The content provider shall not transmit expired object (i.e. objects with an already expired absolute expire time).

6.2.3.1.1.1 Absolute expire times

If the size of the data field is 4 bytes or 6 bytes, then an absolute expire time is defined. The value of the parameter field is coded in the UTC format (see clause 6.2.4.1). It specifies the (absolute) time in UTC when the object expires. The object is not valid anymore after it expired and therefore it shall no longer be presented.

Absolute expire times shall only be used if the expire time is known in advance and if a change to the expire time is considered unlikely. If the content provider wants to signal an expire time, but no absolute expire time is known in advance, then a relative expire time shall be used.

6.2.3.1.1.2 Relative expire times

If the size of the data field is 1 byte, then a relative expire time is defined. While an absolute expire time is known in advance, the relative expire time indicates a validity interval. The interval indicates the maximum time span an object is considered valid after the last time the MOT decoder could verify that this object is still broadcast.

The DataField of this parameter is a one byte value specifying the time the MOT object can still be considered valid starting at the time the MOT decoder no longer receives any segment of the MOT directory (e.g. no more reception or receiver switched off) and therefore the MOT decoder has no reliable knowledge which files are still broadcast. The interval ranges from 2 minutes to 63 days. The interval should be longer than the time between two retransmissions of the MOT directory; the interval shall be longer than the time between the reception of two MOT directory segments. Note that due to reception errors the MOT decoder might miss MOT directory segments.

The encoding of this parameter is shown in figure 26.

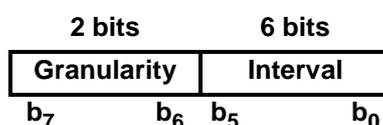


Figure 26: Coding of the MOT parameter Expiration in case of relative expire times

Granularity: this 2-bit field indicates the temporal resolution of the relative expire time.

Interval: this 6-bit unsigned binary number specifies the relative expire time in multiples of the time interval specified by the field Granularity. The value 0 is reserved for future use and shall not be used.

Table 1: Resolution and covered time intervals for relative expire times

Granularity	Temporal resolution	Covered time interval
00	two minutes	from 2 minutes to 126 minutes (approx. 2 hours) (2 minutes, 4 minutes, 6 minutes, 8 minutes, etc.)
01	half hours	from half an hour to 31,5 hours (0,5 hours, 1 hour, 1,5 hours, 2 hours, etc.)
10	two hours	from 2 hours to 5,25 days (2 hours, 4 hours, 6 hours, 8 hours, etc.)
11	days	from 1 day to 63 days (approx. 2 months) (1 day, 2 days, 3 days, 4 days, etc.)

6.2.3.1.1.3 Disabling DefaultExpiration

If the MOT directory extension parameter `DefaultExpiration` is used, but some MOT objects of the MOT data carousel should *never* expire, then the absolute expire time of these MOT objects (MOT header information parameter `Expiration`) shall be set to the maximum value that can be signalled:

- Validity flag = 1.
- MJD: all bits set to 1.
- UTC flag = 0 (short form of absolute time value).
- UTC: set to 23:59.

NOTE: Even if the MOT decoder does no special handling of this specific parameter value, this encoding assures that an MOT object with this absolute expire time will never expire in practice.

6.2.3.1.2 PermitOutdatedVersions

When the MOT decoder notices a change to the data carousel (i.e. it gets a new MOT directory) then this parameter can be used to indicate if an outdated (old) version of an MOT object can be used until the current (new) version of this object is successfully reassembled.

The DataField of this parameter is a single byte. A value of 0 indicates that the MOT decoder shall not use any other version of the MOT object than the one currently broadcast. Any value other than 0 in the current (new) MOT directory indicates that an already available (older) version of an MOT object can be used until the current (new) version of this object is received.

The MOT directory extension parameter `DefaultPermitOutdatedVersions` defined in clause 7.2.4.2 (if present) defines a default value for all MOT objects that do not provide the MOT header information parameter `PermitOutdatedVersions`. The default value defined by `DefaultPermitOutdatedVersions` only applies to an MOT object if no MOT header information parameter `PermitOutdatedVersions` is provided for this MOT object.

If the MOT header information parameter `PermitOutdatedVersions` is provided for an MOT object, then it indicates whether an outdated version of the object can be presented until the current version of the object is received. If `PermitOutdatedVersions` is not provided for the object, then the MOT directory extension parameter `DefaultPermitOutdatedVersions` indicates whether an outdated version of the object can be presented. If neither the MOT header information parameter `PermitOutdatedVersions` is provided for an MOT object nor the MOT directory extension parameter `DefaultPermitOutdatedVersions` is provided as a default for all objects, then the MOT decoder shall not present any outdated version of this object. Note that this is also the behaviour of legacy MOT decoders, since the former revision of the MOT standard did not permit to keep an outdated version of an MOT object.

6.2.3.1.3 UniqueBodyVersion

This parameter is used to uniquely identify a version of an MOT body (identified by its `ContentName`).

The DataField of this MOT parameter is a 32-bit field. The value of two parameters `UniqueBodyVersion` shall be considered the same if both parameters have the same DataField content.

If this MOT parameter is used, then every version of an MOT body must have a unique `UniqueBodyVersion` parameter value. The parameter value can be assigned using whatever scheme. It is not mandatory to increment the parameter value by one for every new version of an object, the only requirement is uniqueness. The MOT decoder shall therefore not use this parameter to "guess" the age of an object's version!

6.2.3.1.4 Priority

The parameter is used to indicate the storage priority, i.e. in case of a "memory full" state only the objects having a high priority should be stored. It indicates the relevance of the content of the particular object for the service, e.g. a home page of a HTML based service has a high priority and should therefore not be deleted first, whereas pictures (e.g. buttons, etc.) are not as important as the home page and hence can be deleted first in case of a memory overflow. The DataField of this parameter carries an 8-bit unsigned binary number. The possible values range from 0 = "highest priority" to 255 = "lowest priority".

Note that usually a caching strategy will take into account the priority value assigned by the content provider as well as user preferences.

6.2.3.1.5 RetransmissionDistance

To support advanced caching of objects in the receiver, this parameter indicates a guaranteed maximum time between two retransmissions of an object. The resolution in the time domain is 1/10 second to allow an exact synchronization, whereas the maximum time which can be indicated reaches up to 1 677 721 seconds (equal approx. 19 days, 10 hours and 2 minutes) for very slow retransmission rates. The DataField of this parameter is encoded as shown in figure 27.



Figure 27: Coding of the MOT parameter RetransmissionDistance

6.2.3.2 MOT conditional access parameters

These parameters are used if conditional access on MOT level is applied to the MOT data.

6.2.3.2.1 CAInfo

The `CAInfo` parameter is used to indicate the scrambling status of individual objects within the data carousel where a service potentially contains both scrambled and unscrambled objects. The syntax of the `CAInfo` parameter is defined in TS 102 367 [8].

Even an MOT decoder that does not support conditional access on MOT level shall check for the MOT parameter `CAInfo` to determine if an MOT body is scrambled. The existence of the MOT parameter `CAInfo` for an MOT object indicates that the body is scrambled. The content of the MOT parameter `CAInfo` does not have to be evaluated by a non conditional access aware MOT decoder, while its presence must be evaluated by every MOT decoder.

6.2.3.2.2 CAReplacementObject

If an object within the data carousel is scrambled and the receiver is unable to unscramble the object, it is desirable for the receiver to be able to present information about how the user may subscribe to the service in order to decrypt the scrambled objects. The `CAReplacementObject` parameter allows this by re-directing the receiver to a replacement object if the receiver is unable to unscramble a given object. The coding of the parameter is exactly the same as the coding of the MOT parameter `ContentName`. When comparing the parameter `CAReplacementObject` with the `ContentName` of an object, the `DataFields` shall be compared byte-by-byte including the character set indicator.

NOTE 1: The replacement object will usually have to have the same object type as the object it replaces. So in case of a BWS, a replacement object for an image should also be an image. The replacement of an HTML page should be an HTML page.

NOTE 2: It might not be necessary to provide `CAReplacementObject` parameters for every scrambled object. But if for instance a part of a BWS is scrambled, then at least all MOT objects that can be reached from unencrypted pages (i.e. the entry pages to the encrypted part of the BWS) should use `CAReplacementObject` parameters to inform the user how he can access the encrypted data part.

User applications such as the Broadcast Web Site permit to link from one object to the other. Usually relative links (e.g. `../images/logo.png`) are used. Relative links require that the presentation engine (e.g. the browser) knows the name of the currently presented object so that all links within this object can be interpreted relative to the name of the current object. The presentation engine (e.g. every browser) will always convert relative links within an object (e.g. an HTML page) to absolute links and always request objects from the MOT decoder by their absolute name.

So if a requested object can not be provided by the MOT decoder because it can not be descrambled, then the MOT decoder must tell the user application decoder that it has to request the replacement object instead. This assures that the user application decoder interprets all links within the replacement object relative to the name of the replacement object (and not relative to the name of the originally requested object). An implementation that just returns the replacement object instead of the originally requested object will not work for user applications that permit relative links between objects!

The replacement object shall not be scrambled.

6.2.3.3 MOT profile identification

These parameters can be used if a user application supports multiple profiles.

6.2.3.3.1 ProfileSubset

The data carousel for a user application carries objects to support more than one user application profile. Additional hinting may be applied by the MOT decoder if it knows which profile a given object is used by. For a receiver conforming to profile *x*, only objects that are relevant to profile *x* receivers need be decoded and stored. The optional `ProfileSubset` parameter allows the content provider to indicate a list of profiles for which any given object is relevant. The `ProfileSubset` parameter data field carries a list of 8-bit unsigned binary number profile ids which identifies *all* of the profiles for which the object is relevant. All profile ids are sorted in ascending order within the list.

This MOT parameter assumes that the user application uses 8-bit unsigned binary number profile ids to signal the profiles supported by the content provider. If the user application uses profile ids of a different size, then this generic parameter can not be used.

If the `ProfileSubset` parameter is not specified for an object in the data carousel, the receiver shall assume that the object may be relevant to all signalled user application profiles.

6.2.4 Coding of parameters

6.2.4.1 Coding of time parameters

Some MOT parameters can signal absolute time values (e.g. `Expiration`).

Absolute times are coded as shown in figure 28 (see also EN 300 401 [1]).

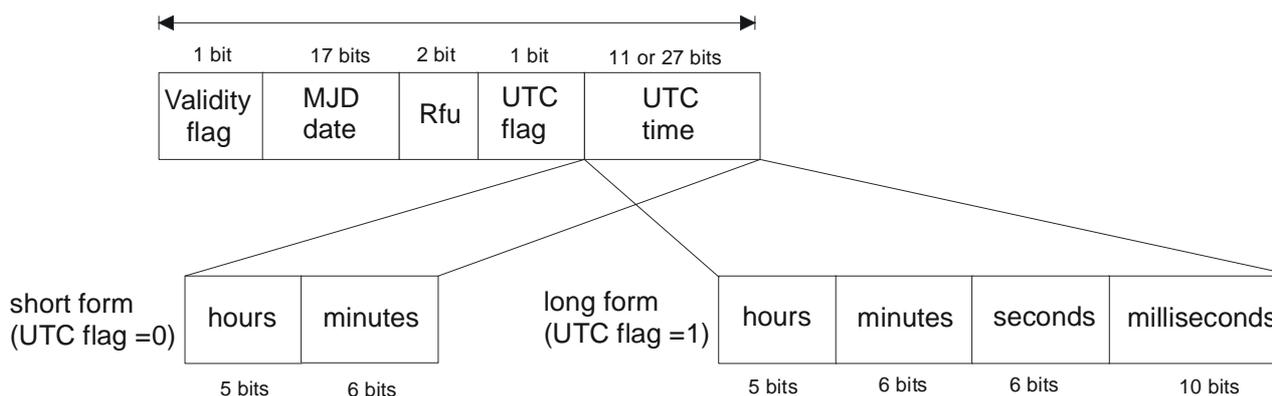


Figure 28: Encoding of absolute time information

Validity flag: This bit is used to indicate whether the time and date information (UTC and MJD) carried in the time parameters is valid or not as follows:

- Validity flag = 0: "Now"; MJD and UTC shall be ignored and be set to 0.
- Validity flag = 1: MJD and UTC are valid.

Depending on the value of the UTC time flag the size of the DataField is 4 bytes (UTC flag = 0) or 6 bytes (UTC flag = 1).

All numeric values (MJD date, hours, minutes, seconds, milliseconds) are coded as unsigned binary numbers.

6.3 List of all MOT parameters in the MOT header extension

The MOT header extension contains transport protocol specific and user application specific parameters. The transport protocol specific parameters are defined in this clause.

The present document also defines the range of Parameter Ids to be used for future extension of the MOT transport protocol and for user application specific parameters.

It is up to the user application to decide which MOT parameters (and also which user application specific parameters, if any) are used and which parameter values are permitted.

A generic MOT decoder will forward all user application specific parameters to the user application decoder, while protocol specific parameters need to be evaluated and handled by the MOT decoder itself.

The following rules apply:

- All mandatory transport specific parameters must be evaluated and handled by the MOT decoder. All optional transport specific parameters may and all unknown transport specific parameters must be ignored by the MOT decoder. For some optional MOT parameters the expected behaviour when ignoring the parameter is explicitly stated (e.g. for `DefaultPermitOutdatedVersions/PermitOutdatedVersions`).
- All user application specific parameters must be handed over to the user application decoder.

General rules for transport specific parameters:

- One MOT parameter is mandatory for both content provider and MOT decoder: `ContentName`.
- Every MOT decoder shall check if an MOT body is compressed (MOT parameter `CompressionType`) or scrambled (MOT parameter `CAInfo`). The MOT decoder does not necessarily (i.e. unless required by the user application) have to be able to decompress or unscramble objects, but it shall be able to identify and discard objects that it can not process.

The following table lists all currently defined MOT parameters carried within the MOT header extension (these parameters refer to a single MOT object); MOT parameters carried in the MOT directory extension (these parameters refer to the data carousel as a whole) are defined in clause 7.2.4. For historical reasons some user application specific parameters are also included. For user application specific parameters not included in this list the meaning of the parameter depends on the user application. It is possible that two user applications use the same `ParameterId` but assign a totally different meaning to it. The decoding of the user application specific parameter always has to take into account which user application provides the data.

Table 2: Coding of extension parameter

Parameter Id b ₅ b ₀	Parameter	Definition	Possible occurrences	Usage mandatory for content provider	Support mandatory for MOT decoders
00 0000	reserved for MOT protocol extensions				
00 0001	PermitOutdatedVersions	6.2.3.1.2	only once	no	no
00 0010 00 0011 00 0100	reserved for MOT protocol extensions				
00 0101	TriggerTime (user application specific parameter)	see [5]	see [5]	see [5]	see [5]
00 0110	reserved for MOT protocol extensions				
00 0111	RetransmissionDistance	6.2.3.1.5	only once	no	no
00 1000	reserved for MOT protocol extensions				
00 1001	Expiration	6.2.3.1.1	only once	no	yes, if receiver provides "MOT caching support"
00 1010	Priority	6.2.3.1.4	only once	no	no
00 1011	Label (user application specific parameter)	see [6]	only once	no	no
00 1100	ContentName	6.2.2.1.1	only once	yes	yes
00 1101	UniqueBodyVersion	6.2.3.1.3	only once	no	no
00 1110 00 1111	reserved for MOT protocol extensions				
01 0000	MimeType	6.2.2.1.2	only once	user application specific	user application specific
01 0001	CompressionType	6.2.2.1.3	only once	yes (if body is compressed)	yes ; every receiver must check if an object is compressed
01 0010 ... 01 1111	reserved for MOT protocol extensions				
10 0000	AdditionalHeader (user application specific parameter)	see [6]	once or several times	see [6]	see [6]
10 0001	ProfileSubset	6.2.3.3.1	only once	no	no
10 0010	reserved for MOT protocol extensions				
10 0011	CAInfo	6.2.3.2.1	only once	yes (if CA is used)	yes ; every receiver must check if an object is scrambled
10 0100	CAReplacementObject	6.2.3.2.2	only once	no	no
10 0101 ... 11 1111	reserved for user application specific parameters				

7 MOT transport modes

The MOT protocol supports two different transport modes:

- MOT header mode;
- MOT directory mode.

MOT header mode is used if there is just one MOT object valid at any time (e.g. user application MOT Slide Show (see TS 101 499 [5]) where a new slide is received, processed, displayed and discarded). In MOT header mode the header information describing the single MOT body is carried as an MOT header.

MOT directory mode is used if file system structures with multiple files are broadcast as a data carousel. All header information of all broadcast MOT objects is combined with parameters describing the set of files. The resulting combined information is called the MOT directory. There is one single MOT directory at any time. A received MOT directory replaces any former MOT directory.

A user application definition shall specify if the user application uses MOT header mode or MOT directory mode.

Some of the features of the MOT directory mode are:

- MOT directory mode permits simple memory management on receiver side since the content provider can explicitly signal which MOT objects are valid and this implicitly also indicates which MOT objects shall be removed on receiver side.
- MOT directory mode can assure consistency of the set of objects. It is possible to indicate which version of an object is to be presented with which version of another object.
- MOT directory mode can be extended to permit persistent caching and delta updating of objects on receiver side. If several MOT objects of a user application are rarely updated, then persistent caching can significantly improve the start-up time of the user application.
- MOT directory mode permits conditional access on MOT level where some objects are accessible on all MOT terminals supporting the user application whereas some other objects are protected by conditional access.
- MOT directory mode combines all header information of all objects in one single MOT directory. Since broadcast of this MOT directory takes some time the MOT directory mode is not well suited for fast changes to the set of broadcast objects.

MOT header mode can be seen as a special case of the MOT directory because it describes the one single MOT object valid at this time. The MOT directory mode can be used to transmit one single object; whereas the MOT header mode is limited to one single file only.

7.1 MOT header mode

This MOT transport mode is used when there is one single valid object at any time. This transport mode is usually used for user applications that decode, process and discard object after object from a stream of objects. An example user application is the MOT Slide Show, see TS 101 499 [5]). If the MOT decoder detects that a new MOT object is being used and the current MOT object is not yet reassembled, then the MOT decoder shall discard the incomplete object and start reassembly of the new MOT object.

In MOT header mode the MOT object consists of one MOT header (containing the header information) describing one MOT body (containing the payload).

The MOT header describing the body shall be sent at least once preceding the body of that object and it can be inserted during the body transmission if required.

The MOT header mode is not intended to carry file system structures (data carousels). Even if a special user application could use the MOT objects coming out of the MOT decoder to rebuild a file system structure on receiver side, the task of the MOT decoder is limited to decoding MOT objects and forwarding them one after the other to the user application decoder.

7.1.1 New object/object update

If a new object is transmitted it will have a new `ContentName` and it will get a new *TransportId*. It will replace any object already stored in the MOT decoder.

If the information carried in the body of an object is updated, a new object has to be transmitted. A body cannot be partly updated since MOT header mode just handles the object as an entity. While the `ContentName` will be the same, the *TransportId* will change. Therefore a new body and a new header have to be transmitted. Header and body segments of the updated object will be transmitted with the new *TransportId*.

MOT provides two methods to signal a change in the header information of an object:

- changing the *TransportId*;
- sending a Header Update.

The first method shall be used, if the header information is to be changed and the transmission of the body is to be continued. Then the transmission of the new object with updated header (but with the same body) is done with a new *TransportId*. If the *TransportId* changes, then the MOT decoder will remove all MOT segments received so far and start reassembly of the new (but actually unchanged) MOT body from scratch.

The second method shall be used, if only the header is changed, but the body is not sent anymore (for example when an object is triggered using a HeaderUpdate that overwrites (or sets) the `TriggerTime`). The MOT decoder shall pass any HeaderUpdate to the user application decoder. It is the task of the user application decoder to check if the HeaderUpdate refers to an object in the user application decoder and it is also the task of the user application decoder to process the header update. See clause 7.1.3.

7.1.2 Management of TransportIds

The *TransportId* field is used to uniquely identify a specific version of an object. In order to minimize the risk of confusion for the MOT decoder when reassembling MOT objects, the content provider has to ensure that *TransportIds* are not re-used until all other available *TransportIds* have been used.

NOTE: The *TransportId* is used solely for the purpose of identifying the object during transport - it has no user application significance whatsoever.

7.1.3 Updating header information/triggering objects

The HeaderUpdate is a specific method of updating the parameters of objects, where both header core and header extension are sent after the entire object (MOT header and MOT body) has already been transmitted. It is used to update MOT parameters (e.g. `TriggerTime`). The header update object shall consist at least of the parameters described hereafter:

ContentName: This parameter is used to link header update to the object to be updated.

ContentType: This parameter shall be set to 0x000101 = MOT Transport.

ContentSubType: This parameter shall be set to 0x0000000000 = UpdateHeader.

BodySize: This field shall be set to zero.

The MOT parameter `ContentName` cannot be replaced during a header update since it is used to link the header update to the object to be updated.

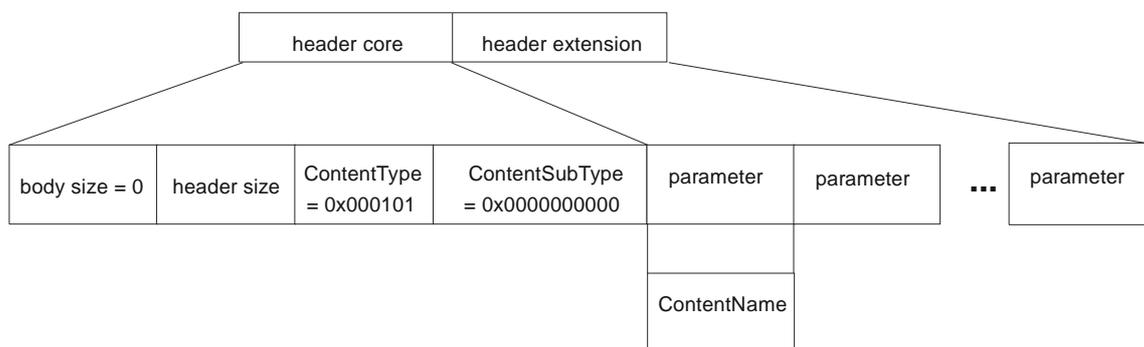


Figure 29: Structure of the header update

7.2 MOT directory mode

7.2.1 Introduction

MOT directory mode is used if file system structures with multiple files are broadcast as a data carousel. All header information of all broadcast MOT objects is combined with parameters describing the overall set of files. The resulting combined information is called the MOT directory. There is one single MOT directory at any time. If a new MOT directory is created it replaces the former MOT directory. This MOT directory describing all MOT objects is sent "in parallel" (see clause 5.3.2.1) to all the MOT bodies it describes.

This clause describes the format of the MOT protocol that provides a management mechanism when broadcasting MOT objects in a data carousel. A data carousel is a delivery system that allows a user application server (the broadcast component of a user application) to present a set of distinct objects to a user application decoder (a program that is run by a receiver) by cyclically repeating the contents of the data carousel. For some user applications the data carousel may complete only a few or a single cycle.

Within the data carousel, the MOT directory is used to provide a complete description of the content of the data carousel, together with sufficient information to find the data for each described object. Version control mechanisms applied both to the objects within the data carousel and the directory itself provide the ability to correctly manage updates to the data carousel with minimum effort and at all times ensure that the correct version of an object is used by the user application.

If a user application requests a particular object, the receiver can easily determine by looking in the directory whether or not the requested object exists within the data carousel and where to find the object data. If the object the user application requests is not yet available in the receiver it may simply wait for the next time that the object is broadcast. If desired, the receiver may optionally implement caching strategies to reduce the latency of accesses by the user application decoder and improve the performance of the data carousel.

7.2.2 Assembly of MOT bodies and MOT directory

MOT transfers objects by dividing both MOT directory and MOT bodies into fixed length segments and then transferring each segment within an MSC data group. In order to reassemble each body, the MOT decoder uses a *TransportId* and a *SegmentNumber* carried in the *Session Header field* of the data group to identify which segment of which body the data group is carrying. The *TransportId* is a unique identifier for a particular version of an object within the data carousel and is also used to provide version management of the data. Whenever the MOT object (MOT header information or MOT body (or the segmentation)) changes, the *TransportId* of the MOT body is also changed.

7.2.3 MOT directory coding

The MOT directory is the table of contents for the MOT data carousel and is the mechanism for controlling access to the objects. Any request for an object can be processed by looking up the object in the MOT directory and using the directory to identify the *TransportId* of the desired object. The directory is also the key to managing version control of objects within the MOT data carousel; if the *TransportId* of the directory changes, the contents of the data carousel should have changed and a simple examination of the directory can identify all the objects that have changed.

The directory contains parameters to describe the entire data carousel together with a list of the required directory information for each object within the data carousel. The structure of the MOT directory is shown in figure 30.

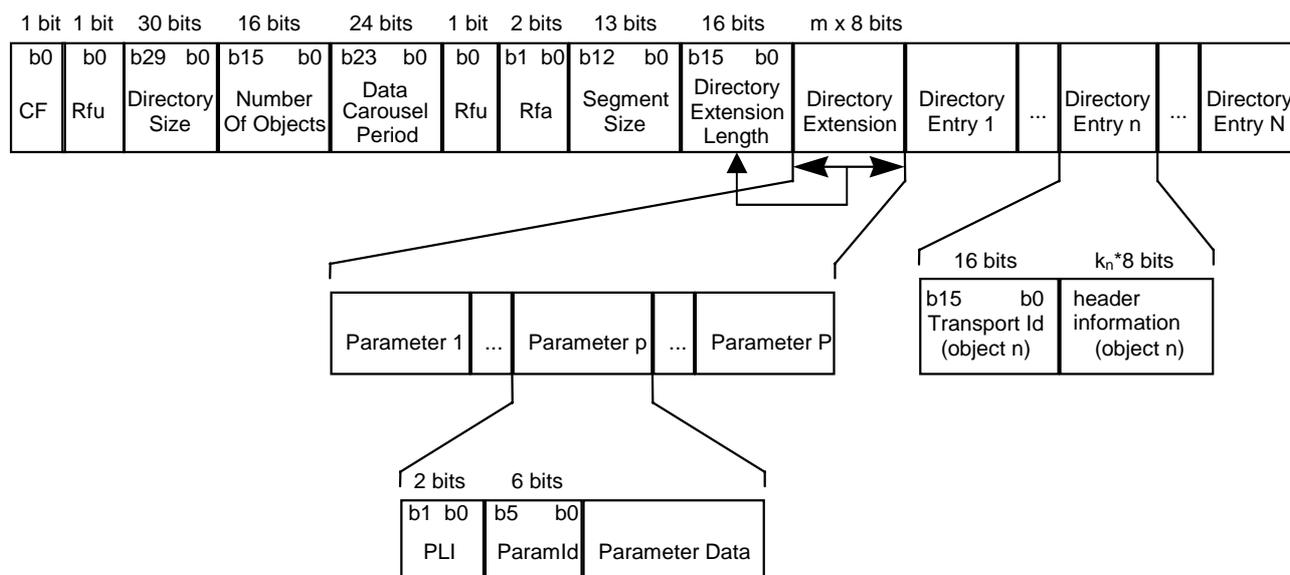


Figure 30: Structure of the MOT directory

CF (CompressionFlag): This bit shall be set to 0.

Rfu: This 1-bit field shall be reserved for future use of the remainder of the structure. The bit shall be set to zero for the currently specified definition of the MOT directory. An MOT decoder has to verify the value of this bit.

DirectorySize: Indicates the total size of the MOT directory in bytes.

NumberOfObjects: Indicates the total number N of objects described by the directory.

DataCarouselPeriod: Indicates the maximum time in tenths of a second for the data carousel to complete a cycle. It is the longest time taken for any object in the data carousel to be retransmitted. A value of 0 shall indicate that the DataCarouselPeriod is undefined.

NOTE: This is the case if the data carousel only makes one turn or the bit-rate changes dynamically.

Rfu: This 1-bit field shall be reserved for future use of the remainder of the structure. The bit shall be set to zero for the currently specified definition of the MOT directory. An MOT decoder has to verify the value of this bit.

Rfa: This 2-bit field shall be reserved for future additions. The bits shall be set to zero until they are defined.

SegmentSize: Indicates the *SegmentSize* in bytes that will be used for the segmentation of objects within the MOT data carousel. A value of 0 indicates that objects within the data carousel may have different segmentation sizes. The last segment of an object may be smaller than this *SegmentSize*.

DirectoryExtensionLength: Indicates the total number of following directory extension bytes.

DirectoryExtension: Carries a list of parameters which are used to describe the entire data carousel. The structure of these parameters is as defined for the MOT header extension parameters (see clauses 6.2 and 7.2.4).

DirectoryEntry: Describes one MOT object. Every DirectoryEntry comprises the TransportId of the MOT object and its MOT header information.

TransportId: Identifies the object to which the following MOT header information refers.

Header information: Carries the header core and header extension of the object. Every object of the data carousel shall be described only once per MOT directory, even if an object is transmitted multiple times during one turn of the data carousel. The coding structure is exactly the same for header information in data groups of type 6 or 7 (i.e. in the MOT directory) and in data groups of type 3 (i.e. in a separate MOT header).

7.2.4 List of all MOT parameters in the MOT directory extension

The directory entries describe individual objects within the data carousel. Parameters in the directory extension describe the overall set of objects, i.e. not an individual object. The structure of these parameters is as defined for the header extension parameters, see clause 6.2.

The directory extension contains transport protocol specific and user application specific parameters. The transport protocol specific parameters are defined in this clause.

This clause also defines the range of Parameter Ids to be used for future extension of the MOT transport protocol and for user application specific parameters.

It is up to the user application to decide which MOT transport specific parameters (and also which user application specific parameters, if any) are used and which parameter values are permitted.

A generic MOT decoder will forward all user application specific parameters to the user application decoder, while transport protocol specific parameters need to be evaluated and handled by the MOT decoder itself.

The following rules apply:

- All mandatory transport specific parameters must be evaluated and handled by the MOT decoder. All optional transport specific parameters may and all unknown transport specific parameters must be ignored by the MOT decoder. For some optional MOT parameters the expected behaviour when ignoring the parameter is explicitly stated (e.g. for `DefaultPermitOutdatedVersions/PermitOutdatedVersions`).
- All user application specific parameters must be handed over to the user application decoder.

The following table lists all currently defined MOT parameters carried within the MOT directory extension (these parameters refer to the data carousel as a whole); MOT parameters carried in the MOT header extension (these parameters refer to a single MOT object) are defined in clause 6.3.

Table 3: Directory extension parameters

Parameter Id b ₅ b ₀	Parameter	Definition	Possible occurrences	Usage mandatory for content provider	Support mandatory for MOT decoders
00 0000	SortedHeaderInformation	7.2.4.1	only once	no	no
00 0001	DefaultPermitOutdatedVersions	7.2.4.2	only once	no	no
00 0010	reserved for MOT protocol extensions				
...					
00 1000					
00 1001	DefaultExpiration	7.2.4.3	only once	no	yes, if receiver provides "MOT caching support"
00 1010	reserved for MOT protocol extensions				
...					
01 1111					
10 0000	reserved for user application specific parameters				
10 0001					
10 0010					
10 0011	reserved for user application specific parameters				
...					
11 1111					

7.2.4.1 SortedHeaderInformation

The parameter is used to signal that the headers within the MOT directory are sorted in ascending order of the `ContentName` parameter within every header information block. The parameter has no `DataField`.

If a receiver gets a new version of the MOT directory it has to compare this new directory with the old version to determine which objects are still valid, which objects are deleted and which objects were updated.

For an efficient implementation it is very helpful if the header information within the MOT directory is already sorted by the transmission side by definition. It will be especially helpful for receivers with limited resources.

If the parameter `SortedHeaderInformation` is used, then the header information within the MOT directory shall be sorted in ascending order of the `ContentName`.

The sorting is done byte by byte, comparing the bytes of the character fields of the `ContentName`. The character set indicator field of the `ContentName` has no impact whatsoever on the sorting. See annex A for a description of the sorting algorithm.

To reduce receiver complexity the content provider should sort the header information and signal it by using this MOT directory extension parameter.

7.2.4.2 DefaultPermitOutdatedVersions

This parameter belongs to MOT functionality "MOT caching support", see clause 8.1.

When the MOT decoder notices a change to the data carousel (i.e. it gets a new MOT directory) then the MOT decoder must be told in the current (new) MOT directory if an old version of an MOT object can be used until the current (new) version of this object is received.

The `DataField` of this parameter is a single byte. A value of 0 indicates that the MOT decoder shall not use any other version of the MOT object than the currently broadcast. Any value other than 0 indicates that an old version of an MOT object can be used until the current (new) version of this object is received.

The MOT directory extension parameter `DefaultPermitOutdatedVersions` defines a default value for all MOT objects that do not provide the MOT parameter `PermitOutdatedVersions`. The default value defined by `DefaultPermitOutdatedVersions` is only used for an MOT object if no MOT parameter `PermitOutdatedVersions` is provided for this MOT object.

If neither the MOT header information parameter `PermitOutdatedVersions` is provided for an MOT object nor the MOT directory extension parameter `DefaultPermitOutdatedVersions` is provided as a default for all objects, then the MOT decoder shall not present any outdated version of this object. Note that this is also the behaviour of legacy MOT decoders, since the former revision of the MOT standard did not permit to keep an outdated version of an MOT object.

7.2.4.3 DefaultExpiration

This parameter belongs to MOT functionality "MOT caching support", see clause 8.1.

The MOT directory extension parameter `DefaultExpiration` is used to indicate a default value that specifies how long an object can still be used by the MOT decoder after reception loss. The coding of this parameter is the same as the encoding for the MOT header information parameter `Expiration`, see clause 6.2.3.1.1.

The MOT directory extension parameter `DefaultExpiration` defines a default value for all MOT objects that do not provide the MOT parameter `Expiration`. The default value defined by `DefaultExpiration` is only used for an MOT object if no MOT parameter `Expiration` is provided for this MOT object.

If neither the MOT header information parameter `Expiration` is provided for an MOT object nor the MOT directory extension parameter `DefaultExpiration` is provided as a default for all objects, then the MOT object never expires.

7.2.5 Segment size of the MOT directory

For MOT objects within an MOT data carousel, the segmentation size may be indicated in the MOT directory (i.e. the parameter `SegmentSize` within the MOT directory core). However, the size of segments for the MOT directory itself cannot be known before the first MOT directory segment is received.

7.2.6 Identification of the MOT directory

The MOT directory is the key to accessing any object within the data carousel and so it shall be possible for an MOT decoder to filter for the directory easily. Within an MOT stream that uses MOT directory mode the following rules apply:

- For each data carousel of objects there shall be **one** MOT directory that describes **all** currently broadcast MOT objects within the data carousel.
- An MOT stream shall contain at most **one** data carousel.

In order to provide easy and effective filtering for the directory, a particular data group type shall be used - Type 6: uncompressed MOT directory or Type 7: compressed MOT directory. To identify the MOT directory, the MOT decoder should filter for the directory by looking for data groups with a data group type 6 or 7. Because there can only ever be one directory within the stream of data groups, this can always be done unambiguously. Clause C.3.4.3 explains how the MOT decoder can acquire the MOT directory even if a user application permits both compressed and uncompressed MOT directory. Once acquired, changes to the directory can always be detected by looking for changes in the *TransportId* of the MOT directory.

7.2.7 Use of the MOT directory mode

7.2.7.1 Segment reception order

The order in which MOT segments are received is unimportant for the MOT decoder - the *SegmentNumber* and *TransportId* fields of each segment allow accurate reconstruction of the MOT directory (and of each MOT body) regardless of when the individual segments are received.

7.2.7.2 Service acquisition

The key to acquiring a service broadcast in a data carousel is reception of the MOT directory. Once this has been received, the complete structure and contents of the data carousel is known, even if the data for the objects themselves has not yet been received. If the scope of the data carousel is known, the receiver has all the information it requires to process requests for an object from a user application - it knows whether or not a requested object exists within the data carousel and how to identify the object when it is received.

The MOT decoder can always determine the correct contents of the data carousel by examining the current directory - there is no need for an MOT decoder to have any knowledge about previously broadcast information in order to correctly decode the current data carousel.

7.2.7.3 Version control

The use of a data carousel implies a user application data set that is essentially static - it should be unlikely that the data carried in the data carousel will change rapidly. However, the data may well need to change and it is important that an MOT decoder is able to detect when the data carousel has changed so that it can properly manage any stored data, if applicable.

Each object in the data carousel has a *TransportId* assigned to it that is carried both in the body segments (for verification and identification) and in the MOT directory (for data carousel management). If any object in the data carousel is changed (segmentation, header information or body), a new *TransportId* shall be assigned to the object. This requires a change to the directory and so the *TransportId* of the directory shall also be changed to reflect this - therefore any change to the data carousel can be detected merely by checking for changes in the *TransportId* of the MOT directory.

7.2.7.4 Allocation of TransportIds

The *TransportId* field is used to uniquely identify a specific version of an object within the data carousel. In order to minimize the risk of confusion for the MOT decoder when rapid updates are taking place, the content provider has to ensure that *TransportIds* are not re-used until all other available *TransportIds* have been used. If a data carousel update occurs the transmission side shall not re-use *TransportIds* of objects within the old MOT directory for new or updated objects in the new MOT directory.

NOTE: The *TransportId* is used solely for the purpose of identifying the object during transport - it has no user application significance whatsoever.

7.2.7.5 Prioritizing objects within the data carousel

Because the transmission order of objects (and also their segments) within the data carousel is unimportant, it follows that objects which have more significance than others to the user application may be repeated multiple times within one turn of the data carousel, in order that the acquisition time for these objects is minimized. In particular, the directory may be treated in this way, as it is central for the MOT decoder to being able to access the objects within the data carousel.

NOTE: The MOT parameter `RetransmissionDistance` may be used to indicate the guaranteed maximum time until individual objects appear again within the overall data carousel, as this may differ from the period of the entire data carousel (which is defined as the longest repetition period for any object in the data carousel).

7.2.7.6 Managing updates to the data carousel

When the data carousel is changed there is no requirement to complete either the current cycle of the data carousel or the current object.

If the data carousel is changed the next cycle of the data carousel should then start with the new or updated objects being broadcast first.

The transmission side shall also assure that all objects within the data carousel are broadcast even if a small portion of the data carousel is updated/added more frequently than one data carousel cycle. It is important that a change to the data carousel does not completely restart the transmission cycle. To give an example:

- If the data carousel cycle is five minutes and some small object is changed every minute, then the data carousel shall not restart from scratch and shall not start repeating the first minute of the data carousel without ever broadcasting the data scheduled for the remaining four minutes.

As soon as an MOT decoder detects a change to the directory, it shall use the information in the new directory to determine whether or not any previously stored MOT body is still valid.

See annex F for more detailed information how the content provider has to manage changes to the MOT data carousel.

7.2.7.7 MOT decoder behaviour in case no data is received for a long time

If the MOT decoder receives no data for more than one hour, then the MOT decoder shall completely stop processing MOT body segments and it shall no longer reassemble MOT bodies. Once reception is restored, then the MOT decoder has to first successfully reassemble the current MOT directory. It then has to check if all MOT bodies that are currently reassembled still use the same *TransportId*. Only then the MOT decoder is permitted to continue reassembly of MOT bodies.

The above behaviour is required to assure that an MOT decoder that did not get data for a very long time does not assume that MOT body segments it receives still refer to the very same version of a body.

EXAMPLE: The MOT reassembly unit is ordered to reassemble an MOT body A with *TransportId* X. Before the MOT body can be completely reassembled, the MOT decoder loses reception. For a long time the MOT decoder receives no MOT segments at all. After some time the content provider does not broadcast object A any more and later on re-uses the *TransportId* X for another object B.

If reception is now restored, then it might happen that the MOT body segments with *TransportId* X (now referring to MOT object B) are received before the current MOT directory is completely reassembled. If the MOT reassembly unit waiting for MOT body segments with *TransportId* X (the outdated object A) now processes the newly received MOT body segments as usual and tries to complete reassembly of what it believes to be object A, then a mix of MOT body segments of object A and object B will result. Since there is no error detection on MOT entity level, the MOT reassembly unit can not detect such inconsistencies and might thus forward a "successfully" reassembled MOT body with *TransportId* X to the object management.

As soon as the MOT decoder has completely reassembled the current MOT directory, it can detect that the *TransportId* now refers to another MOT object and restart reassembly of object B from scratch, but at this time already erroneous data could have been forwarded to the user application decoder.

To avoid the inconsistencies explained in the above example, it is necessary that after a reasonable time without reception the MOT decoder first verifies that the *TransportIds* of the MOT body segments still refer to the same MOT objects before it continues reassembly of MOT bodies.

NOTE 1: The content provider has to ensure that *TransportIds* are not re-used until all other available *TransportIds* have been used. Therefore it will usually take very long until a *TransportId* is reused. However, the receiver could have been without reception for a very long time or in the meantime there could have been a change of the service configuration (e.g. a completely new data carousel (same user application type) is now broadcast in the same data channel) and the receiver now gets data that uses the same *TransportIds*, but for completely different objects.

To detect "reception loss", the MOT decoder keeps track of the last time the MOT decoder received any segment of the current MOT directory (i.e. of the MOT directory currently used by the object management). If no segment of the current MOT directory is received for more than one hour, then the MOT decoder shall stop reassembly and shall first reassembly and evaluate the current MOT directory before it continues reassembly of MOT bodies.

Reassembly of the MOT directory shall not use MOT directory segments older than one hour.

NOTE 2: Keeping track of the last time an MOT segment of the current MOT directory was received is also needed for the evaluation of relative expire time. The parameter that holds the last time any segment of the current MOT directory was received can thus be used for the evaluation of relative expire times and also to detect "reception loss".

NOTE 3: Receivers with advanced reassembly units (see also clause C.3.4.1.1) do not have to discard MOT body segments once reassembly is stopped. They are permitted to put the MOT body segments into the "segment buffer" and they may replay the buffered MOT body segments as soon as the current MOT directory is received. Note that the "segment buffer" will discard all MOT body segments that are older than one hour.

7.2.8 MOT directory compression

The MOT directory contains all `ContentNames` describing the directory structure of the current data carousel. Data structures containing long "file names" are usually very efficiently compressible.

MOT directory compression is used to reduce the size of the MOT directory, allowing a much more economic and efficient usage of the (limited) available bandwidth. The repetition rate of the very important MOT directory may be increased and/or more useful content may be sent over the broadcast channel.

The standard (uncompressed) MOT directory indicates the length of its complete MOT entity at the beginning of the entity. The compressed MOT directory is also preceded by a data field that indicates the size of the MOT entity at the beginning of the entity. Assembly of the compressed MOT directory and the uncompressed MOT directory is thus exactly the same. However, the compressed MOT directory uses the data group type 7 during transfer. The only additional action that must be performed for compressed MOT directories is to de-compress its content (`CompressedMOTDirectoryData`) before it is evaluated.

1 bit	1 bit	30 bit	8 bit	2 bit	30 bit	C x 8 bit
Compression Flag	rfu	EntitySize	CompressionId	rfu	Uncompressed DataLength	Compressed MOTDirectoryData

Figure 31: Structure of a compressed MOT directory

CompressionFlag: This bit shall be set to 1.

rfu: This bit is reserved for future use of the remainder of the structure. This bit shall be set to 0 until it is defined. An MOT decoder has to verify the value of this bit.

EntitySize: This 30-bit field, coded as an unsigned binary number, indicates the total size of the entity in bytes (9 bytes header + length of CompressedMOTDirectoryData (C)).

CompressionId: This 8-bit field coded as an unsigned binary number indicates the compressed data format. For the values of this field see clause 6.2.2.1.3.

rfu: This 2-bit field bit is reserved for future use of the remainder of the structure. These bits shall be set to 0 until they are defined. An MOT decoder has to verify the value of these bits.

UncompressedDataLength: Length in bytes of the standard (uncompressed) MOT directory (after decompression of the CompressedMOTDirectoryData field). This information is provided to simplify memory management for the MOT decoder.

CompressedMOTDirectoryData: The standard MOT directory (see clause 7.2.3) in a compressed form (using the compressed data format specified by parameter CompressionId).

The first bit of the MOT entity (the CompressionFlag) indicates to the MOT decoder if the MOT entity carries a standard (uncompressed) MOT directory or a compressed MOT directory.

Data group type 6 is used to carry the uncompressed MOT directory. Data group type 7 is used to carry the compressed MOT directory. If both compressed and uncompressed MOT directory (describing the very same data carousel) are sent in parallel, then both shall use the same *TransportId*.

All MOT decoders that support MOT directory compression shall also support the standard (uncompressed) MOT directory (that is carried in data group type 6). Clause C.3.4.3 indicates how an MOT decoder that supports MOT directory compression can easily reassemble a compressed and/or uncompressed MOT directory.

It is up to the user application to decide if MOT directory compression is permitted and which compressed data formats are permitted.

It is recommended that all new user applications permit MOT directory compression.

8 MOT functionality

8.1 MOT caching support (MOT directory mode only)

Before MOT caching is explained in detail, this clause will first outline at a very simple MOT decoder without any caching. Such a very simple MOT decoder will automatically reassemble the MOT directory but not reassemble MOT objects until the user application decoder requests an object. Such an MOT decoder might even discard every received object once it was forwarded to the user application decoder. For every request by the user application decoder such an MOT decoder has to wait until the requested MOT object is broadcast for the next time and until all its segments have been successfully received. If such an MOT decoder is able to inform the user application decoder if data used by the user application decoder got updated by the content provider then such an MOT decoder could be perfectly compliant to the MOT standard.

It is clear that the user experience of such an MOT decoder would be unacceptable because the access times of the objects would be abysmal: since the data rate within the broadcast channels are relatively low compared to the data that is broadcast it can take dozens of seconds or even minutes until all data is broadcast. If an object is broadcast once per minute an MOT decoder that needs to reassemble this object might have to wait a minute (or even multiple minutes in case of reception errors) until the object is successfully received.

To assure that the access time to MOT objects (i.e. the time a user application decoder (and thus the user) has to wait to get the requested MOT object) is as short as possible, all MOT decoders will use caching.

The aim of caching is an improved access time to the objects of a user application; caching shall not change the appearance of the user application. Caching, especially persistent caching where data is stored on permanent memory such as a hard disk, shall always present the data as the content provider indicates in the broadcast channel. Care has to be taken to assure that the user application never presents outdated or inconsistent data (i.e. invalid data). In the case of the very simple MOT decoder outlined above no data that can no longer be received (i.e. because it is no longer broadcast by the content provider or because there is no reception) will be presented, thus the very simple MOT decoder assures that no outdated content is presented.

For caching MOT decoders the consistency and validity of data that is so crucial for the content provider is assured by a set of MOT transport specific parameters.

Caching implies different tasks:

- *Object reassembly*: The MOT decoder will try to use all successfully received MOT segments so that all broadcast MOT objects are reassembled and available to the user application decoder as soon as possible. Most MOT decoders will start reassembly of all MOT objects in parallel as soon as the MOT directory is received for the first time. More advanced MOT decoders will even use MOT body segments that were received before the MOT directory could be completely reassembled (see clause C.3.4.1).
- *Object validity*: the MOT decoder has to assure that the user application decoder gets only MOT objects that the content provider wants to be available at the time the object is requested by the user application decoder.
- *Object management*: the MOT decoder must do memory management in case the data carousel contains more objects than the MOT decoder is able to store in its memory. In this case the MOT decoder will try to keep those MOT objects in its cache that are "most likely" or "most frequently" requested by the user application decoder in order to minimize the average access time.

To support caching MOT decoders the MOT protocol provides parameters that help the MOT decoder to determine if a cached object can be consider valid (i.e. forwarded to the user application decoder upon request). Other MOT parameters support the memory management of the MOT decoder.

Note that the content provider might use just some of the MOT parameters defined for MOT caching. He might for instance decide not to provide MOT parameters `RetransmissionDistance` or `Priority`.

The following tables list all MOT parameters used for MOT caching support.

Table 4: Caching support parameters in the MOT header information

Parameter Id b ₅ b ₀	Parameter	Definition	Possible occurrences	Usage mandatory for content provider	Support mandatory for MOT decoders
00 0001	PermitOutdatedVersions	6.2.3.1.2	only once	no	no
00 0111	RetransmissionDistance	6.2.3.1.5	only once	no	no
00 1001	Expiration	6.2.3.1.1	only once	no	yes, if receiver provides "MOT caching support"
00 1010	Priority	6.2.3.1.4	only once	no	No
00 1101	UniqueBodyVersion	6.2.3.1.3	only once	no	No

Table 5: Caching support parameters in the MOT directory

Parameter Id b ₅ b ₀	Parameter	Definition	Possible occurrences	Usage mandatory for content provider	Support mandatory for MOT decoders
00 0001	DefaultPermitOutdatedVersions	7.2.4.2	only once	no	no
00 1001	DefaultExpiration	7.2.4.3	only once	no	yes, if receiver provides "MOT caching support"

The following clauses give additional details to the above given tasks and they also describe the MOT parameters that support these tasks.

8.1.1 Object reassembly

The object reassembly unit should try to reassemble all MOT objects as soon as possible. Preferably the MOT decoder should be able to decode all MOT objects in parallel, i.e. use all MOT segments that are successfully received.

To improve the acquisition time of the broadcast MOT objects, the MOT decoder should also be able to use MOT segments received before an MOT directory could be first reassembled (at MOT decoder start-up) as well as MOT segments of new/updated MOT objects received before the updated MOT directory could be successfully reassembled, see clause C.3.4.1.

8.1.2 Object validity

Assuring object validity means that only those objects are presented to the user, that the content provider wants to be presented at this point in time.

If the user application decoder requests an object, then the standard behaviour of an MOT decoder will be to check if the requested MOT object is part of the current MOT directory. The MOT decoder will then check if the current version of the requested MOT object is already reassembled and if the MOT objects is valid; in this case the MOT decoder will forward this MOT object to the user application decoder.

MOT parameters will assist the MOT decoder answering the following questions:

- 1) How does the MOT decoder know if an MOT object is part of the current directory?
- 2) How does the MOT decoder know if an available MOT body can be forwarded to the user application decoder?

Answer to question 1: As long as the MOT decoder receives MOT directory segments all the time, it is easy to determine if an MOT object is part of the current MOT directory. As soon as we take into account that there might be no or just little reception, this issue gets more complex. Receivers might experience bad reception, receivers might get outside of the reception area (e.g. if the receiver is in a subway train or too far away from the transmitter) or receivers might be tuned to another ensemble. In all these cases the MOT decoder is unable to reassemble the currently broadcast MOT directory and may even be unable to detect if the MOT directory has changed at all.

Note that to detect a change to the MOT directory the MOT decoder just needs to receive a single MOT directory segment. By checking its *TransportId* the MOT decoder will detect if there was a change to the MOT directory. Successful reassembly of the full updated MOT directory is needed in order to be able to tell which MOT objects were affected.

Since the MOT decoder might have no chance to verify if an MOT object is still broadcast the content provider can indicate how long an MOT object can still be considered valid after reception is lost. Absolute and relative expire times assure that no outdated content is presented to the user.

NOTE: There is a fundamental difference between object management (list of objects provided by the MOT directory) and object expiration (provided by the MOT parameters `Expiration` and `DefaultExpiration`). As long as the MOT decoder receives the MOT directory, it will reliably know which objects are valid. Since the transmission side is not permitted to transmit outdated objects, every object listed in the MOT directory is implicitly valid. But if the MOT decoder no longer receives the MOT directory, the MOT decoder has no means to tell if an object is still part of the MOT directory. Therefore the MOT parameters `Expiration` and `DefaultExpiration` can be used to tell the MOT decoder how long an object can be considered valid once the MOT directory is no longer received (and the MOT decoder can no longer verify that the object is still valid).

MOT objects that are no longer part of the MOT directory shall be removed from the MOT decoder cache. Expired objects on the other hand shall not be *presented* any more, they do not necessarily have to be removed, see clause C.3.5.1.

For a description of absolute and relative expire time parameters see clause 8.1.2.1. The support of the MOT parameters `Expiration` and `DefaultExpiration` is mandatory for all MOT decoders that use caching.

Answer to question 2: Two MOT parameters help the MOT decoder to determine if an already available MOT body can be forwarded to the user application decoder. The MOT standard requires that the current version of an MOT object is forwarded to the user application decoder, but the content provider can permit the MOT decoder to use an older version of an object until the current version is successfully received.

If the header information or the segmentation of an MOT object is changed then a change of the `TransportId` of this object is required. The MOT parameter `UniqueBodyVersion` indicates if the MOT body of an MOT object (identified by its `ContentName`) is the same as the formerly received MOT body of this MOT object. If the MOT body is the same then no reassembly of the MOT body is necessary and the already available MOT body can be used. This MOT parameter is especially useful for MOT decoders that use persistent caching. For details regarding this MOT parameter see clause 8.1.2.2.

If MOT objects are updated, an updated version of the MOT directory will be broadcast. As soon as the MOT decoder successfully reassembled the updated MOT directory it will detect which MOT objects were updated. From the moment the MOT decoder detected a new version of an MOT object it could still take some time until the new version of this MOT object is successfully reassembled, see figure 32.

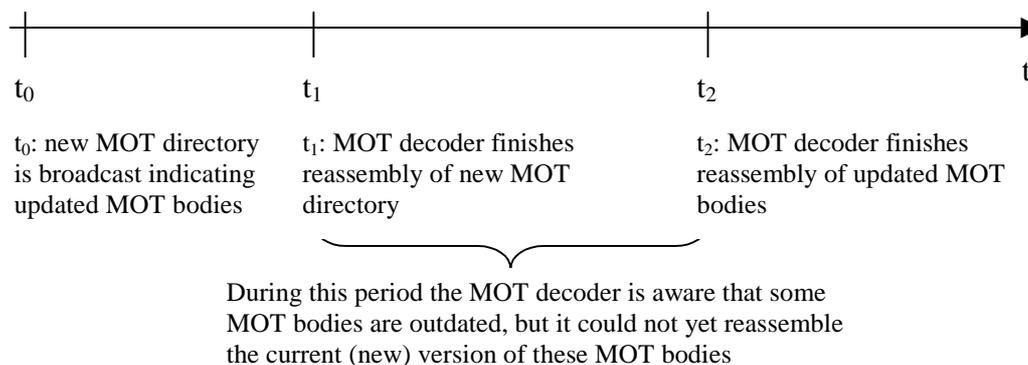


Figure 32: Delay between detecting changes to the data carousel and receiving the current (new) data

The MOT directory extension parameter `DefaultPermitOutdatedVersions` and the MOT header information parameter `PermitOutdatedVersions` indicate if an already available older version of an MOT object may be used until the current version of the MOT body has been received, see clause 8.1.2.3.

It is recommended that MOT decoders support both parameters `PermitOutdatedVersions` and `UniqueBodyVersion`.

8.1.2.1 MOT expire time handling

If the MOT decoder provides "MOT caching support", then support of the MOT parameters `Expiration` and `DefaultExpiration` is mandatory.

The MOT parameters `Expiration` and `DefaultExpiration` are used to indicate an absolute or a relative expire time of an MOT object. An absolute expire time is used, if the (absolute) time when the object expires is known in advance (e.g. for an advertisement).

A relative expire time is used to indicate the maximum time span an object is considered valid after the last time the MOT decoder could verify that this object is still broadcast. This parameter can for instance be used for news articles that should be presented as long as they are broadcast. If reception is lost, then they can still be presented; but they will not be presented if reception is lost for more than the indicated period specified by the content provider.

A relative expire time is for instance used for regularly updated data applications, where it is not known in advance when an object expires (e.g. when a traffic information should be removed). Absolute trigger times could theoretically be used and adjusted at every update to the data carousel. But every change to MOT header information causes a change to the `TransportId`, therefore such behaviour would cause removal-and-reassembly of the object's body on simple MOT decoders. Simple MOT decoders assume that in case of a change to the `TransportId` also the object's body is changed and therefore they reassemble the (same) MOT body from scratch.

EXAMPLE: A traffic information data application with an update interval of e.g. 5 minutes will usually set the relative expire time to a multiple of the update interval of the data carousel; e.g. to 15 minutes. So traffic information can still be browsed in the underground, but if the MOT decoder can not validate for more than 15 minutes that the traffic information is still broadcast (i.e. the MOT decoder receives no MOT directory), then the traffic information will no longer be presented.

Annex E gives an example of the use of relative expire times.

Every time an MOT object is requested by the user application decoder, the MOT decoder shall evaluate the `Expiration/DefaultExpiration` parameters as follows:

- First the MOT decoder checks if the MOT object has the MOT header information parameter `Expiration`. If it has, its value is used and this evaluation process ends.
- Then the MOT decoder checks if the MOT directory extension parameter `DefaultExpiration` is available. If it is, its value is used and this evaluation process ends.
- If neither the MOT header information `Expiration` for the object nor a default value in the MOT directory extension parameter `DefaultExpiration` is available, then the object never expires.

If the above evaluation yields an absolute expire time (i.e. the size of the data field is 4 bytes or 6 bytes), then the MOT decoder compares the given absolute expire time with the current time. If the object is not yet expired, the MOT decoder will forward the MOT object to the user application decoder.

If the above evaluation yields a relative expire time (i.e. the size of the data field is 1 byte), then the MOT decoder adds the given relative expire time to the last time the MOT decoder got a segment of the current MOT directory. The MOT decoder compares the resulting time with the current time. If the object is not yet expired, the MOT decoder will forward the MOT object to the user application decoder.

A battery powered real time clock is required for persistent caching so that expire times can be checked even if there is no DAB reception at all.

The MOT decoder is not required to delete an expired object (see also annex C); but it shall no longer provide this object to the user application decoder.

If the content provider explicitly gives expire times, then no MOT decoder shall forward expired MOT objects to the user application decoder. It is not the receiver manufacturer to decide if outdated content should be preferred over no content at all. This is the content provider's prerogative.

NOTE: For some user applications the content provider might choose to explicitly indicate the authoring date and time inside the transmitted content (e.g. a BWS might include a time stamp for every news article). In this case the user can see how newsworthy the content really is. However, if the content provider does not want outdated content to be presented to the user even in case of reception loss, he must be able to rely on the automatic and reliable object expiration done by the MOT decoder. It is clear that it can not be the user's duty to verify the time stamp of every information received via DAB. That is the reason why the handling of object expiration is mandatory for caching MOT decoders.

8.1.2.2 Unique MOT body version

If the MOT decoder gets a new MOT directory it checks for every MOT object if it uses the same *TransportId* in the new MOT directory. If it does, then the MOT decoder knows that it is the very same object (header information and body) as it was broadcast before.

If the *TransportId* of the MOT object differs, this could be caused by:

- a change to the header information (e.g. a changed MOT parameter *Expiration*);
- a change to the MOT body;
- differing segmentation size.

It is very helpful for the MOT decoder to know if the change of the *TransportId* was made because of some change to the MOT header information or if also the MOT body was changed. In the latter case the MOT decoder has to re-assemble the current (new) version of the MOT body. This will take some time and thus increase the access time for the user application.

This parameter *UniqueBodyVersion* helps the MOT decoder to determine if an already available (i.e. already received) MOT body still has the very same content even if the *TransportId* for this MOT object has been changed in the current (new) MOT directory. If this parameter is used by the content provider and evaluated by the MOT decoders then changes to MOT header information can be made without implicitly invalidating the MOT body of an MOT object (and thus causing reassembly of the MOT body).

This parameter *UniqueBodyVersion* uniquely identifies a version of the body of an MOT object (identified by a *ContentName*). If the body of an MOT object is changed, the parameter *UniqueBodyVersion* (if broadcast) of the MOT object shall be changed. The MOT encoder shall never use the same value for the parameter *UniqueBodyVersion* (if broadcast) for an MOT object unless it refers to the very same body content for this MOT object (identified by a *ContentName*). Because *UniqueBodyVersion* uniquely identifies a version of an MOT body, this parameter is also used to support persistent caching of MOT objects where MOT decoders need to reliably determine if an MOT body in the persistent cache is the same as the MOT body currently broadcast.

To determine if an already available version of an MOT body has exactly the same content as the currently broadcast version of the MOT object the MOT decoder shall compare the parameter *UniqueBodyVersion* as well as the parameter *BodySize*. If both parameters have the same value for old and new MOT object version, then the MOT decoder may safely assume that the body content is exactly the same.

Two MOT objects (i.e. having two different *ContentNames*) that have the same value for the parameter *UniqueBodyVersion* do NOT have to have the same body content. The parameter *UniqueBodyVersion* shall only be used to compare two versions of one MOT object (i.e. the *ContentName* of the two versions must be the same).

If the parameter *UniqueBodyVersion* is not available for both versions of an MOT object then the MOT decoder can not reliably determine if two versions have the same body content. In this case the MOT decoder shall assume that the body content differs when it detects that the *TransportId* was changed.

An MOT decoder that does not support this parameter *UniqueBodyVersion* shall also assume that the body content differs when it detects that the *TransportId* was changed (this is the default behaviour of the MOT decoder anyway).

Examples how the values for the parameter *UniqueBodyVersion* could be managed by the content provider:

- the parameter is treated as a 32-bit unsigned binary version number. For every new version of an MOT object the MOT encoder increments the version number for this object;

- the content provider uses a 32-bit data field that holds the date/time when the object was created. It must be assured that no two different versions of an object are created having the same time stamp. If a version of an object has exactly the same content as an older version, its MOT parameter `UniqueBodyVersion` could be set to the same value the older version had.

The parameter `UniqueBodyVersion` is used exclusively to determine if two versions of an object have the same body content. This parameter shall not be used to derive an order (e.g. age of an object's version)!

Every receiver that supports the parameter `UniqueBodyVersion` shall also support the parameters `Expiration` and `DefaultExpiration` which are mandatory for "MOT caching support".

8.1.2.3 Temporarily using outdated MOT bodies

The MOT directory indicates the versions of all MOT objects within the data carousel so that the MOT decoder can assure a consistent set of objects at all times.

Usually a consistent set of objects is required. E.g. if a news content provider broadcasts an HTML page with an inline image and later updates both HTML page and inline image (i.e. keeping the same `ContentName` for the HTML page and the inline image) with other news, then the MOT directory assures that the MOT decoder knows which version of the HTML object and which version of the inline image belong together and the MOT decoder can assure that news will never be presented with the wrong inline image.

If consistency is required, then the MOT decoder is not permitted to provide a different version of an MOT object than indicated in the MOT directory (i.e. an earlier received MOT object with the same `ContentName`). The MOT decoder has to wait until the current version of the object is received. So if the HTML page with the current football scores is not yet received the MOT decoder is not permitted to present the old object during the time until the new object is received. The MOT decoder must therefore remove an object from its cache as soon as it detects that a different version of an object (with the same `ContentName`) is broadcast.

The MOT header information parameter `PermitOutdatedVersions` as well as the MOT directory extension parameter `DefaultPermitOutdatedVersions` use a data field of one byte. A value of 0 indicates that the MOT decoder may not use any other version of an object than the current (new) one. If the current (new) version of the MOT object is not yet available when requested by the user application decoder, then the MOT decoder shall NEVER use an old version of this MOT object. Any value other than 0 means that the MOT decoder may use ANY older version of an object (provided that this version of the object has not expired) until the current (new) version of the object is received.

The parameter `DefaultPermitOutdatedVersions` used within the new MOT directory defines the default behaviour for the objects within the data carousel. An MOT header information parameter `PermitOutdatedVersions` can set a different behaviour for a single MOT object.

For every updated MOT object the MOT decoder shall evaluate the `PermitOutdatedVersions/DefaultPermitOutdatedVersions` parameters as follows:

- First the MOT decoder checks if the MOT object has the MOT header information parameter `PermitOutdatedVersions` (in the current (new) MOT directory). If it has, its value is used and this evaluation process ends.
- Then the MOT decoder checks if the MOT directory extension parameter `DefaultPermitOutdatedVersions` is available (in the current (new) MOT directory). If it is, its value is used and this evaluation process ends.
- If neither the MOT header information parameter `PermitOutdatedVersions` for the object nor a default value in the MOT directory extension parameter `DefaultPermitOutdatedVersions` is available, then a value of 0 (it is not permitted to use outdated versions of any object) shall be used.

If the above evaluation yields a value other than 0 then the MOT decoder is permitted to provide older and not yet expired versions of MOT objects (i.e. having the same `ContentName`) from its cache until the current version of the MOT object is successfully received. The current (new) version of the MOT object shall then replace the older version as soon as it is successfully received.

The MOT decoder shall never provide a version of an object that is already expired, even if the MOT decoder is permitted to provide outdated versions of an object!

Use of a value other than 0 (outdated versions of an object may be used) for the parameter `PermitOutdatedVersions` implies that the content provider uses the same `ContentName` for an MOT object if the MOT object is updated. If the content of an MOT object can not be considered an update of an MOT object then the MOT object shall get a new `ContentName`. Since MOT decoders might use persistent caching the content provider must assure that a `ContentName` is not reused unless it can be considered an update for ALL earlier (and not yet expired) version of this MOT object.

The default behaviour of MOT directory mode assures consistency. If neither the MOT directory extension parameter `DefaultPermitOutdatedVersions` nor the MOT header information parameter `PermitOutdatedVersions` is available for an MOT object then the MOT decoder shall NOT use an older version of the object.

If an MOT decoder does not evaluate the `DefaultPermitOutdatedVersions/PermitOutdatedVersions` parameters it shall never present older versions of an MOT object.

Since little or no consistency requirements will usually provide a better user experience the content provider should try to set up his user application so that consistency requirements are as little as possible (i.e. whenever possible, the content provider should permit to temporarily use an outdated version of an MOT object).

If MOT consistency is not required (i.e. outdated versions of objects may temporarily be used), then care has to be taken with respect to receivers that use persistent caching. The use of expire times is highly recommended for content providers, see clause 8.1.2.1.

If an outdated version of an MOT object may temporarily be used, then the ENTIRE MOT OBJECT shall be used until the current (new) version of the MOT object is successfully reassembled. No MOT decoder shall use an outdated version of the MOT body together with the current (new) version of the MOT header information! See clause C.3.5.1.1 for implementation tips.

Every receiver that supports the parameters `DefaultPermitOutdatedVersions/PermitOutdatedVersions` shall also support the parameter `Expiration` and `DefaultExpiration`; the latter two parameters are mandatory for "MOT caching support".

8.1.3 Object management

If the MOT decoder does not have enough cache memory to hold all MOT objects within the current MOT directory it must use some strategy to decide which objects it will keep in its memory and which MOT object to remove in case memory is needed.

The MOT decoder will try to keep those MOT objects in its cache that are "most likely" requested by the user application decoder in order to minimize the average access time.

The MOT parameter `Priority` indicates the content provider's default storage priority. An MOT decoder should initially prefer MOT objects with high priority to MOT objects with lower priority. Usually a caching strategy will take into account the priority as assigned by the content provider as well as user preferences. This means that if the memory management can get an indication of the user's preferences or a list of objects that will most likely be needed by the user from the user application decoder (e.g. in case of a Broadcast Website all the HTML pages that can be reached from the currently displayed HTML page), then this information will most likely have stronger impact on the memory management strategy than the (static) information provided by the content provider.

The MOT parameter `RetransmissionDistance` indicates the maximum time between two retransmissions of an object. The MOT decoder can take this value as an indication how long it will take to reassemble an MOT object (taking into account that several retransmissions of an object might be needed in case of reception problems). Note that a strategy that prefers to cache rarely broadcast objects to objects that are retransmitted every few seconds might have the undesired side effect that those objects that the content provider considers to be the most important ones (hence they are broadcast so frequently) have the longest access times.

Note that the parameter `RetransmissionDistance` indicates the (maximum) time between two retransmissions. The actual time interval might be much shorter. It is also important to note that this parameter does not indicate the time interval between reception of the MOT directory and reception of the MOT body. It indicates the time between two MOT body retransmissions.

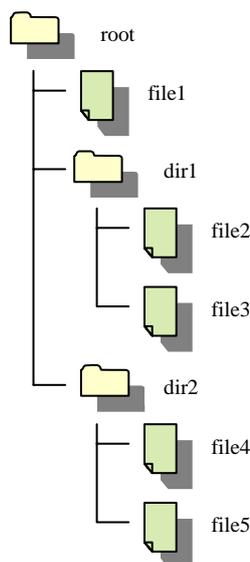
8.2 Transfer of directory structures using MOT

User applications may need to transfer a set of files to the receivers in a certain directory structure. For these user applications to work on the receiver it is necessary to carry the complete directory structure from the user application provider side to the receiver side.

A user application that wants to use the MOT functionality "transfer of directory structures" shall indicate this in the user application definition. The interface between MOT decoder and user application shall then assure that the directory structure of the files on content provider side can be rebuilt on receiver side.

NOTE 1: A generic MOT decoder will not use the MOT parameter `ContentName` as (part of) the filename where it stores the MOT body (see clause 6.2.2.1.1). The user application decoder will use the appropriate interface of the MOT decoder to access the broadcast directory structure.

An example for a directory structure is given below.



For transmission the complete path of every object in the directory structure is signalled in the MOT parameter `ContentName`. The absolute path consists of <n> directory path components and the basename of the file. Between two directory path components (if any) as well as between the last directory path component and the basename of the file the character slash ("/") is used as delimiter.

The following rules shall be observed for the `ContentName`:

- No `ContentName` shall have a leading "/".
- The use of ".", "..", "?" and "*" as a path name component or as a basename is not allowed.
- The character "\" shall not be used. The directory hierarchy delimiter is the "/" character.
- No references to physical or logical devices are allowed (for example "C:", "/dev/ttyS1").
- For an end-to-end system, where the MOT directory transfer is the transportation means, system dependent characters encodings in the `ContentName` can be used.
- The `Character Set Indicator` should be set to the character set the user application uses. This will most probably be ISO/IEC 8859-1 [10], also known as ISO-latin1 (character set indicator value 4) or ISO/IEC 10646 [11], UTF8 encoding (character set indicator value 15). Character set indicator values besides 15 or the values 0 to 4 shall not be used. Neither the MOT decoder nor the interface between the MOT decoder and the user application shall do any character set conversion. The MOT decoder shall completely ignore the character set indicator.

Depending on the user application additional rules may apply.

EXAMPLE: If files for a computer game for a Windows system are transported, then all characters permitted by Windows systems may be permitted. If a user application should run on as many receiver platforms as possible, then the permitted characters might be much more restricted.

NOTE 2: If a user application uses characters within the `ContentName` that are not permitted by the MOT standard or the user application definition, then the transmission side may convert these characters to permitted characters (e.g., replace "\" by "%5c"). Many user applications will use references from one file to another file (e.g. a "playlist" will reference audio files, an executable might reference software libraries or entry pages). In such a case also the references must be adjusted.

Examples for valid `ContentNames`:

```
root/dir1/file2
images/logo.jpg
news.xml
many/subdirectories/for/this/file.txt
```

Examples for invalid `ContentNames`:

```
root\dir1\file3           (the character "\" shall not be used as directory delimiter)
root/dir2/./dir1/file    (".." shall not be used)
images/logo/           (no basename is given)
/images                 (a leading "/" is not permitted)
com1:                   (a logical device on Windows™ systems)
```

Other MOT parameters than the `ContentName` can be used for each object in the carousel. The user application description shall define which parameters are relevant and their possible values. The following list mentions some MOT parameters that the user application might want to consider using:

- `MimeType`: indicates the type of the object. This parameter is useful if the user application needs an explicit indication of an object's type (e.g., Broadcast website). Other user applications might use the file extension or other means to identify an object's type. The user application must specify if it distinguishes different types of objects. If it does, then the user application must specify how the user application decoder can determine the type of an object.
- `CompressionType`: indicates that an object is compressed and which compression algorithm has been used. The user application will define whether the MOT objects can be compressed and which algorithms are supported. Evaluation of this parameter is mandatory for the MOT decoder. An MOT decoder that cannot decompress an object shall discard it.

NOTE 3: Some implementations might decide to decompress some objects inside the user application decoder instead of the MOT decoder. In such a case, if neither the MOT decoder nor the user application decoder can decompress an object, then the MOT decoder shall discard it.

- `ProfileSubset`: allows the service provider to indicate a list of profiles for which the given object is relevant. This parameter will be useful if the user application supports multiple profiles and some files are not used by all profiles.
- `CAInfo`: this parameter shall always be checked to determine if the body is scrambled, even if the receiver does not support conditional access. If the MOT decoder is unable to descramble an object it shall discard the object.
- Additional user application specific MOT parameters might be needed by such user applications.

Caching support and conditional access are optional for both the content provider and the receiver. If Caching is supported by the receiver, then at least the parameters `Expiration` and `DefaultExpiration` shall be evaluated by the MOT decoder.

Transfer of directory structure uses the MOT Directory.

Annex A (normative): Comparing ContentNames

This "compare" function is needed if the MOT directory extension parameter `SortedHeaderInformation` inside the MOT directory signals that its header information is sorted in ascending order of the parameter `ContentName` within every header information.

When sorting the header information a function assuring the same sorting order as the following function shall be used to compare two `ContentNames`. The same function will also be used inside the MOT decoder to compare an old MOT directory with the currently received one.

The reference function is specified in C code:

```
/*
compare:
the pointers cn1_ptr and cn2_ptr point to the "character field" (sequence of bytes) of the
ContentName (i.e. the character set indicator and the rfa bits
are not compared and does not influence sorting)
returns a negative number if c1 < c2;
returns 0 if both ContentNames are the same
returns a positive number if c1 > c2
*/
int compare (const unsigned char *cn1_ptr, unsigned int cn1_length,
             const unsigned char *cn2_ptr, unsigned int cn2_length)
{
    int ii, diff;
    int min = cn1_length;
    if (cn2_length < min) min = cn2_length;
    for (ii = 0; ii < min; ii++)
    {
        diff = cn1_ptr[ii] - cn2_ptr[ii];
        if (diff != 0) return diff;
    }
    if (cn1_length < cn2_length) return -1;
    if (cn1_length > cn2_length) return 1;
    /* both strings are equal */
    return 0;
}
```

Annex B (informative): User application definitions and MOT

Every user application that uses MOT has to define if it uses:

- MOT header mode; or
- MOT directory mode.

If the user application needs user application specific MOT parameters or MOT directory extension parameters, it has to define their meaning and encoding and assign a `ParameterId`. Different user applications might assign totally different meanings to the same `ParameterId`. The meaning and encoding of user application specific parameters depends on the user application being decoded.

If MOT directory mode is used and MOT directory compression is permitted by the user application definition, then the user application also has to define which compression methods for the MOT directory are supported.

For every parameter the user applications has to specify if:

- the parameter is optional or mandatory for the content/service provider;
- the parameter is optional or mandatory for the MOT decoder/user application decoder;
- the parameter can occur once or multiple times per MOT object;
- there are some restrictions on the permitted values of the parameter (e.g. a restriction on the length or the permitted characters inside a `ContentName` or on the permitted compression data formats of the parameter `CompressionType`).

The gzip compression data format supports multiple gzip window sizes that determine the memory requirements for decompression. If gzip compression is permitted for MOT directory compression and/or MOT body compression, then it is also necessary to specify the list of permitted gzip window sizes.

If MOT header mode is used the user application has to indicate if `HeaderUpdates` are permitted and which MOT parameters will be updated.

For some MOT functionality available in MOT directory mode such as "caching" or "conditional access on MOT level", more than one parameter is needed. So usually the complete functionality including all its parameters will be optional or mandatory. The user application should not explicitly indicate which caching or conditional access parameters are to be supported; it should just mention the functionality.

It is recommended for every user application definition to permit the MOT functionality "caching" - if not mandatory, "caching" support should at least be optional for both content provider and MOT decoder.

It is recommended for every user application definition to permit the use of conditional access on MOT level. The minimum requirement for every MOT decoder is that it shall always check if the MOT parameter `CAInfo` is signalled for an MOT object. Even if the MOT decoder does not support conditional access on MOT level, it does know that the presence of the MOT parameter indicates a scrambled MOT body.

Every user application definition should explicitly state that all MOT decoders shall check for the MOT parameter `CompressionType`. Every MOT decoder should at least be able to detect and discard all MOT bodies it can not uncompress.

Like the MOT parameters `CAInfo` and `CompressionType`, all user application specific parameters that change the encoding of the MOT body (i.e. the MOT body can not be processed if this user application specific parameter is signalled) also need to be mandatory for the MOT decoder/user application decoder.

User applications that foresee fast changes to the data carousel should require collecting of MOT body segments whose *TransportIds* are not (yet) listed in the current MOT directory (see clause C.3.4.1 and annex F).

In addition to the MOT related settings every user application definition also has to define how the user application is signalled in the FIC (FIG0/13). A user application identifier needs to be defined and the user application specific data (if any).

If a user application supports different profiles, then the list of user application profiles supported by the content provider should be carried in the User Application Data field in FIG0/13. If the MOT parameter `ProfileSubset` should be used, then the profile ids of all supported profiles must be 8-bit unsigned binary numbers that should be sorted in ascending order within the User Application Data field.

Annex C (informative): Model of an MOT decoder and its interfaces

The model describes the functionality of the MOT decoder on different levels including the interfaces to a DAB receiver (providing a stream of MSC data groups) and the terminal (running an MOT based user application decoder), see also figure 1. Real implementations may be quite different, optimized according user application specific needs and receiver design constraints etc.

The data flow up to the user application decoder and the interfaces between all the levels described in this clause are summarized in figure C.1.

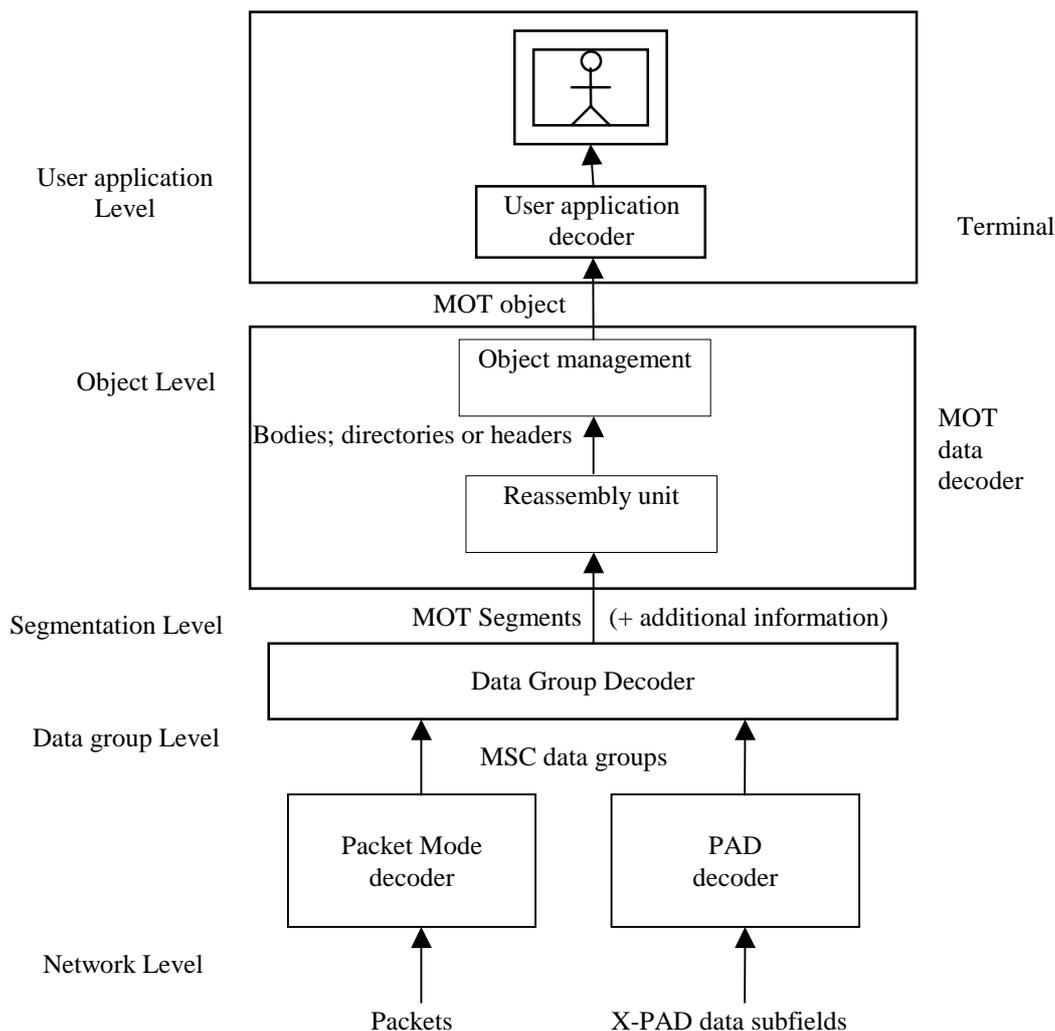


Figure C.1: General description of a MOT decoder and its interfaces

Figure C.2 explains the functionality of the MOT directory mode decoder in detail.

Note that the object management also performs caching of the MOT bodies.

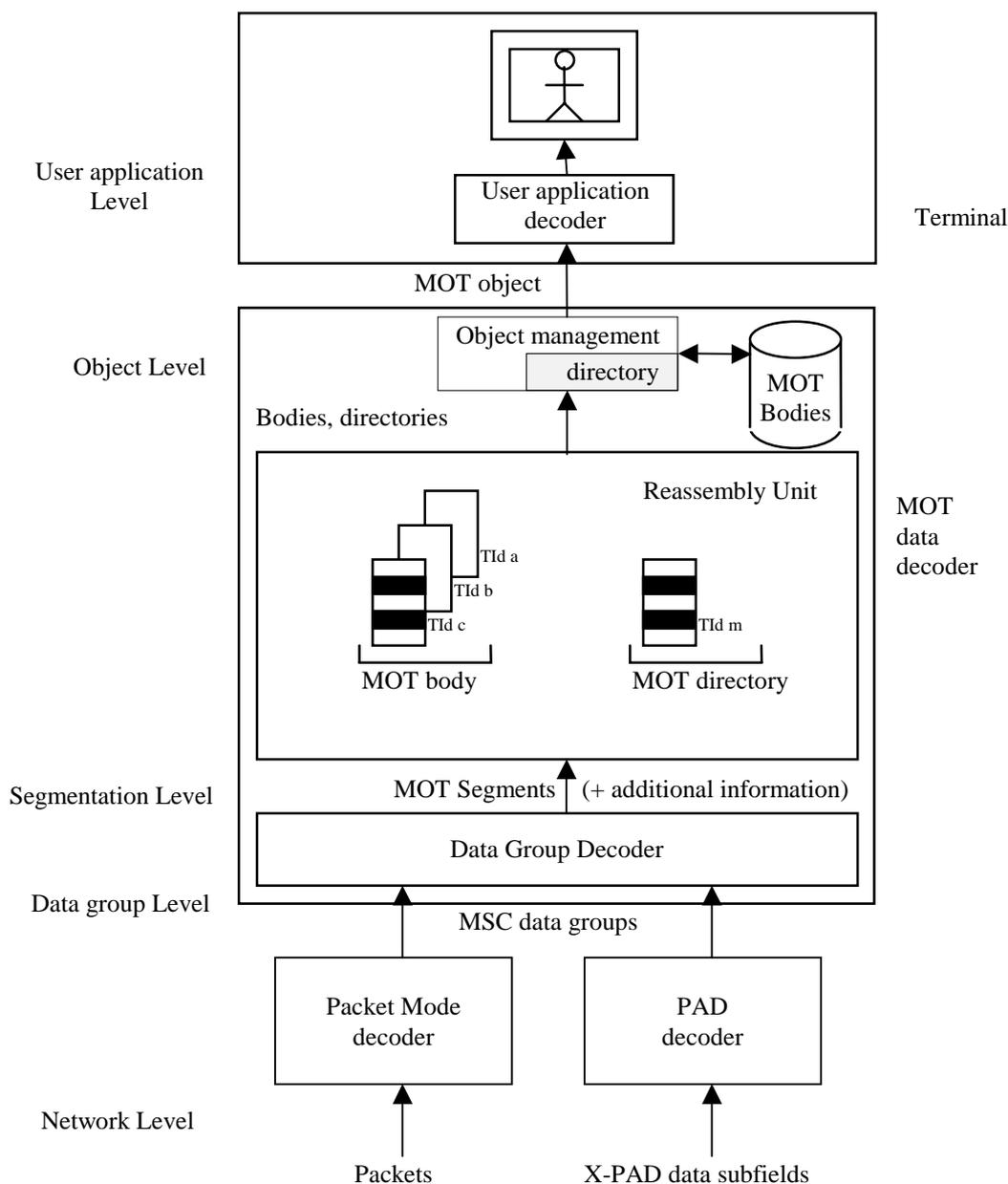


Figure C.2: General description of a MOT directory mode decoder and its interfaces

Figure C.3 explains the functionality of the MOT header mode decoder in detail.

Note that object management is very simple. Incoming MOT objects are just forwarded to the user application decoder as soon as they are fully reassembled from their MOT segments. There is no MOT object caching in MOT header mode.

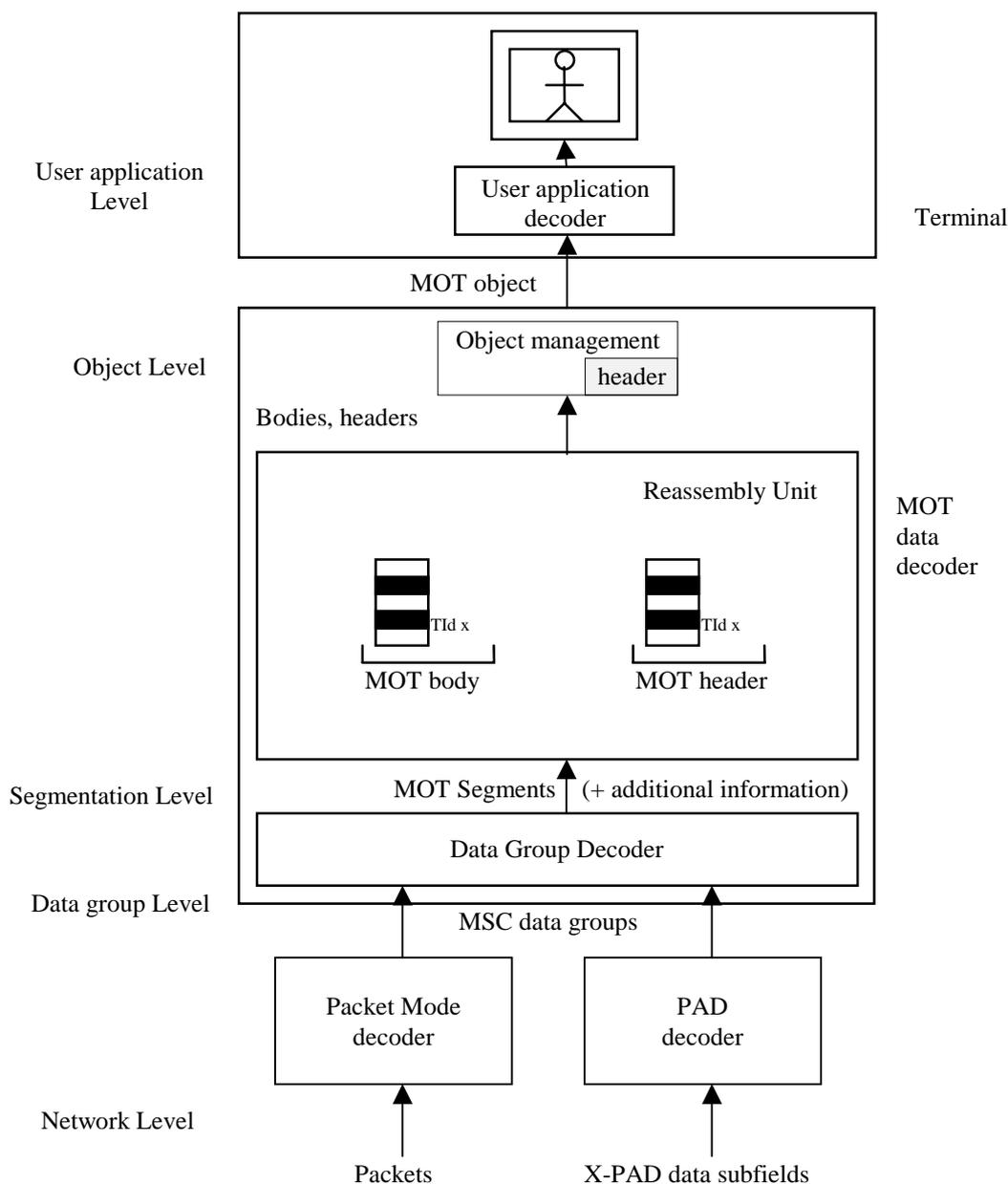


Figure C.3: General description of a MOT header mode decoder and its interfaces

C.1 Network level

At the network level packets and/or X-PAD data subfields are processed as described in clause 5.2 and complete MSC data groups are passed to the data group level.

In packet mode, the packet address has to be used to identify a particular service component within a subchannel. The packets are collected by taking into account the packet Continuity index. The length of the data group is derived from the First/Last flag and the Useful data length in the packet headers. The validity of each packet is verified by the evaluation of the packet CRC.

In X-PAD mode, the X-PAD application type is used to identify all X-PAD subfields belonging to the user application. The MSC data group length is derived from the DataGroupLengthIndicator (X-PAD application type 1) immediately preceding the start of the MSC data group.

C.2 MSC Data group level

The validity of each single MSC data group is verified by the evaluation of the MSC data group CRC. An MOT decoder needs to decode only data group types 1, 3, 4, 5, 6 and 7. If the MOT decoder does not support conditional access on MOT level, then it only needs to decode data group types 3, 4, 6 and 7. Data groups of type 7 are only decoded if the MOT decoder supports MOT directory compression. Data groups with other Data group types are discarded. The MSC data group data field contains a complete segment (including the segmentation header with *RepetitionCount* and *SegmentSize*). The corresponding *SegmentNumber* and the *TransportId* are provided by the session header of the data group. A CRC checked segment, together with its corresponding Segmentation header, Data group Type, *TransportId* and *SegmentNumber* (see clause 5.1 for more details) will be passed to a re-assembly unit working at the segmentation level.

C.3 Segmentation and object level

The reassembly unit reassembles segments with the same *TransportId*. The reassembly unit processes data groups with Data group type 3 (MOT header), type 4 (MOT body), type 5 (scrambled MOT body and CA parameters), type 6 (uncompressed MOT directory), type 7 (compressed MOT directory) and type 1 (CA messages).

C.3.1 General description of the MOT decoder

The MOT decoder consists of two parts:

- The reassembly unit reassembles MOT headers, MOT bodies and the MOT directory.
- The object management controls the reassembly unit, stores the received objects and handles requests by the user application.

In this general description two operation modes of an MOT decoder are described:

- The MOT header mode: in this mode MOT header and body are processed (one MOT object at a time).
- The MOT directory mode: in this mode the MOT directory and multiple bodies are processed in parallel.

Both reassembly unit and object management unit are in the same mode, either in MOT header mode or in MOT directory mode. The mode is determined by the user application.

C.3.2 The reassembly unit

The functionality of the reassembly unit depends on its operation mode:

- MOT header mode: in this mode MOT headers and bodies are reassembled (Data group types 3 and 4).
- MOT directory mode: in this mode the MOT directory and bodies are reassembled (Data group types 6/7 and 4; for CA also Data group types 1 and 5).

The reassembly unit continuously evaluates the incoming data groups carrying MOT segments. It shall be prepared that in case of MOT directory mode several objects are transmitted applying interleaving, so that they are to be decoded in parallel. It is not required by the reassembly unit to evaluate the data group Repetition index or the *RepetitionCount* of the segmentation header.

C.3.2.1 MOT directory mode

The MOT directory is reassembled and forwarded to the object management. If the MOT directory is forwarded, its *TransportId* is stored inside the reassembly unit. From now on all MOT directory segments transported with this *TransportId* are discarded, since this MOT directory is already successfully reassembled. If an MOT directory segment is received with a different *TransportId* this means that the MOT directory is updated and therefore the new MOT directory is reassembled and forwarded and its *TransportId* stored.

The object management orders the reassembly unit to reassemble MOT bodies. It is up to the object management to assure that there is enough memory to store these bodies and still reassemble and forward a new MOT directory.

An MOT decoder that does not support conditional access will not order reassembly of scrambled MOT bodies.

C.3.2.2 MOT header mode

When an MOT header is forwarded (to the object management), its *TransportId* is kept inside the reassembly unit. From now on all MOT header segments with this *TransportId* can be ignored, because they are already known by the object management. If a new *TransportId* is detected (data group types 3 or 4), then the MOT header mode decoder removes all MOT header and body segments received so far (no matter if they belong to an already forwarded MOT object or not) and reassembles the MOT header of the new MOT object.

The object management orders the reassembly unit to reassemble the MOT body. It is up to the object management to assure that there is enough memory to store the body and still reassemble and forward a new MOT header.

C.3.2.3 Segmentation of MOT bodies

Reassembly of MOT bodies is independent from the mode the reassembly unit is in. If the object management requests a body, the reassembly unit gets a request indicating which bodies (one body in case of MOT header mode; possibly multiple bodies in case of MOT directory mode) are to be reassembled. This request will include the *TransportId*, the size of the bodies, an indication if the body is scrambled (MOT directory mode only) and may include also the *SegmentSize* (if given in the MOT directory). The reassembly unit can thus allocate memory for the requested bodies.

The *TransportId* is not only used to reassemble all segments of an MOT header/MOT directory or an MOT body of an MOT object, but also to establish the link between header information and body and to link them to the related CA messages, if applicable. If the reassembly unit is ordered to reassemble a body, then it shall collect data groups of type 4 (unscrambled body) or data groups of type 5 and 1 (scrambled body/CA messages) of the indicated *TransportId*.

C.3.3 The object management unit

The object management stores objects and permits the user application to request objects, e.g. by their *ContentName*. It is the object management that evaluates the header information of reassembled MOT headers or MOT directories (depending on the operation mode) and orders reassembly of MOT bodies.

According to the model described in this clause, it is the object management that orders the reassembly unit to decode the MOT bodies. The reassembly unit will only reassemble MOT bodies requested by the object management.

The functionality of the object management depends on its operation mode.

C.3.3.1 MOT directory mode

In case of MOT directory mode the object management can order the reassembly unit to reassemble multiple MOT bodies. Depending on the memory capacity, caching strategies (e.g. object priority) and CA support of the MOT decoder, the object management will determine the MOT bodies that are to be reassembled. The reassembly unit will then process all MOT body segments of the requested MOT bodies as they are incoming, either sequentially one after the other or interleaved, in parallel. Since the object management unit already knows the size of the bodies from the MOT directory, it can assure that there is enough memory for the reassembly unit to hold all bodies reassembled in parallel, and a new MOT directory (the size of the MOT directory is not known in advance).

In MOT directory mode, the reassembly unit will inform the object management after the MOT directory is completely reassembled. The object management decompresses (if applicable) and processes the MOT directory and then orders the reassembly unit either to reassemble all incoming bodies or- in case of memory shortage - to select the specific objects that are to be reassembled, e.g. by starting to request first the entry object(s) of the data carousel or objects with a high *Priority* parameter until all objects have been received. If there is not enough memory to hold all objects within the data carousel, the object management uses a caching strategy that determines which objects should be stored. A very simple caching strategy would require the reassembly unit to reassemble only the body of the object that is currently requested by the user application and to store these bodies. A more advanced strategy will try to reassemble bodies before they are requested by the user application and thus reduce access time.

If a new directory is forwarded by the reassembly unit, the object management first checks if it is a compressed MOT directory. If the MOT directory is compressed the object management de-compresses it. Then the object management compares the old directory and the new one and removes all objects from its cache that are no longer signalled within the directory (i.e. their `ContentNames` are no longer present in the new MOT directory) or that have been updated (i.e. a different *TransportId* is used for the same `ContentName`). It will then remove the objects that are no longer present in the MOT directory and order the reassembly unit to reassemble new and updated MOT objects.

Comparing old and new MOT directory can be dramatically simplified if the header information within the MOT directory is sorted by the `ContentNames`, see clause 7.2.4.1 for details.

The object management tries to reduce the object access time and thus includes some caching strategy.

Although the terminal will allocate the greatest part of its memory to the storage of the objects, it shall be prepared for situations, where the size and number of objects exceeds the memory resources. In such cases, the object management can make use of additional information on the relevance and availability of every object, i.e. by evaluating the caching parameters in the header information of the MOT object for optimal use of memory.

An example showing the actions of the MOT decoder after it receives a new MOT directory can be found in annex D.

C.3.3.2 MOT header mode

In MOT header mode the reassembly unit continuously checks for MOT headers and forwards these to the object management.

A received MOT header forwarded from the reassembly unit replaces any former MOT header in the object management since there is just one MOT object at any time.

The object management will order the reassembly unit to reassemble the single MOT body described by the current MOT header. Since the object management unit already knows the size of the body from the MOT header, it can assure that there is enough memory for the reassembly unit to hold the MOT body and a new MOT header.

When the MOT body has been successfully and completely assembled in the reassembly unit, it is passed over to the object management in the MOT decoder. The object management will then automatically forward the received MOT object (header information and body) to the user application decoder.

After successful reassembly of an MOT object, incoming MOT segments with the same *TransportId* are discarded.

The MOT decoder shall pass any `HeaderUpdate` to the user application decoder. It is the task of the user application decoder to check if the `HeaderUpdate` refers to an object in the user application decoder and it is also the task of the user application decoder to apply the header update.

C.3.4 Advanced MOT reassembly units

C.3.4.1 Collecting MOT body segments whose *TransportId* is not described in the MOT directory

The MOT object management tells the reassembly unit which MOT bodies have to be reassembled and which MOT bodies do not have to be reassembled. To do this, the object management will provide a list of all the *TransportIds* within its current MOT directory and indicate for every *TransportId* if the MOT reassembly unit has to process MOT segments having this *TransportId* (i.e. reassemble the MOT body) or if the reassembly unit has to discard MOT segments having this *TransportId*.

If the *TransportId* of a received MOT body segment is in the list provided by the MOT object management, then the reassembly unit will process the MOT body segment as usual (i.e. the MOT segment is used to reassemble an MOT body or it is discarded).

This clause only covers MOT body segments whose *TransportId* the MOT object management did not provide (i.e. because the MOT object management does not yet have the current MOT directory and therefore the object management does not know the *TransportIds* of new or updated MOT objects).

This clause not only explains how to improve start-up of the MOT decoder and robustness in case of reception errors during updates of the MOT directory. The mechanisms also permit fast changes to the data carousel (see annex F).

C.3.4.1.1 Start-up of the MOT directory mode decoder

When the MOT decoder starts up it will take a while until the MOT directory is successfully received and forwarded to the object management. If the reassembly unit discards all body segments until the MOT directory is successfully received then the start of reassembly of MOT bodies can be significantly delayed.

An advanced MOT decoder can thus include a small storage in the reassembly unit ("segment buffer") that collects all MOT body segments received between start up of the reassembly unit and the order to collect certain (or all) MOT bodies from the object management unit. If the object management unit tells the reassembly unit which MOT bodies it should collect then the reassembly unit "replays" the already stored MOT body segments, flushes the storage and from then on processes the MOT body segments as they are received.

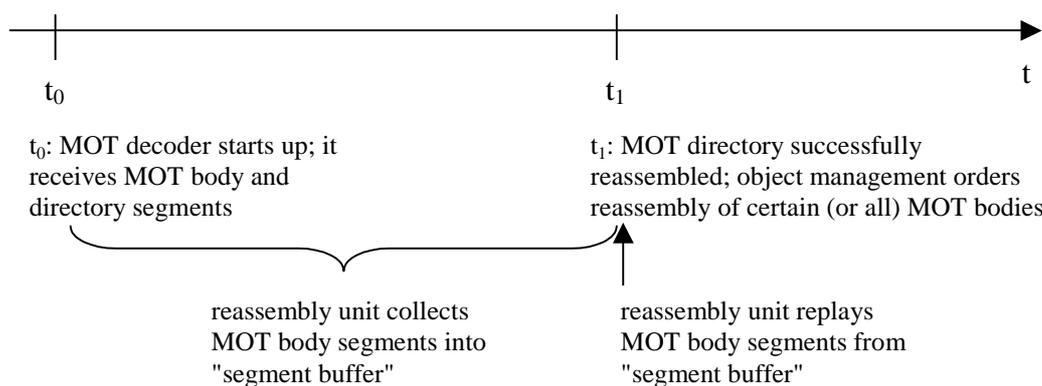


Figure C.4: Improved start-up of MOT decoder

Additional information regarding the collection of MOT body segments can be found in clause C.3.4.1.3.

C.3.4.1.2 Updates to the MOT directory

If changes to the data carousel are made, then the content of the MOT directory changes and therefore a new *TransportId* is assigned to the MOT directory. After the changes to the data carousel the new MOT directory and the new and updated as well as the unchanged MOT bodies will be broadcast. The MOT decoder will continuously check every received MOT directory segment and notice that MOT directory segments with a new *TransportId* are received.

Due to reception errors it could take quite a while until the new MOT directory is successfully reassembled and forwarded to the object management. If the reassembly unit discards all body segments until the new MOT directory is successfully received then the start of reassembly of updated and new MOT bodies can be significantly delayed.

An advanced MOT decoder will thus have an interface that permits the object management to tell the reassembly unit for all *TransportIds* within the current MOT directory what *TransportIds* the object management is interested in and which *TransportIds* it is NOT interested in. If MOT body segments are received with *TransportIds* that are not contained in the current MOT directory (i.e. not at all mentioned by the object management) then the reassembly unit should assume that a change to the data carousel has taken place. It should then collect the MOT body segments into the "segment buffer". When the next MOT directory is successfully forwarded to the object management and the object management unit tells the reassembly unit which MOT bodies it should collect, then the reassembly unit "replays" the already stored MOT body segments, flushes the "segment buffer" and from then on processes the MOT body segments as they are received.

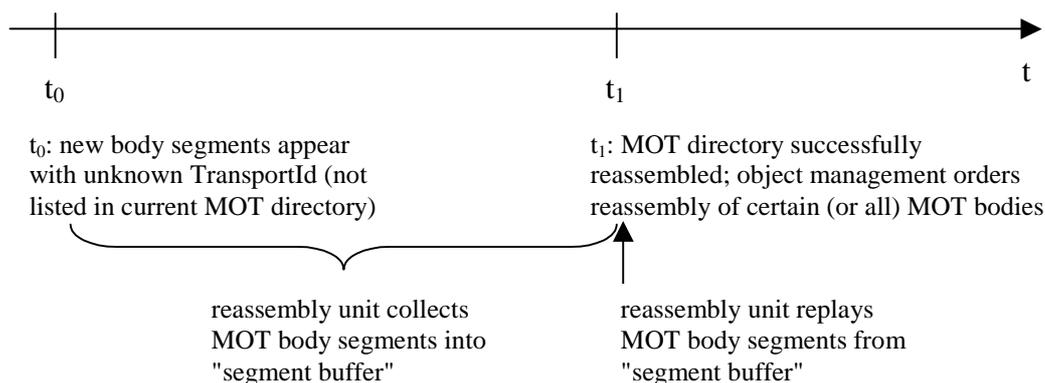


Figure C.5: Improved performance in case of updates to the MOT directory

Additional information regarding the collection of MOT body segments can be found in clause C.3.4.1.3.

C.3.4.1.3 Collecting MOT body segments

The MOT reassembly unit automatically reassembles the MOT directory, but it is the task of the object management to decide which MOT bodies should be reassembled at any given time. The MOT object management therefore provides a list of ALL *TransportIds* within the current MOT directory to the MOT reassembly unit. For every *TransportId* the object management will indicate if its MOT body has to be reassembled or not.

On MOT decoder start-up, this list will be empty (the object management simply does not know the current MOT directory on start-up). Whenever the reassembly unit successfully rebuilds a new (the current) MOT directory, this MOT directory will be forwarded to the object management. The object management will decide which MOT bodies are to be reassembled and indicate this to the reassembly unit.

Whenever an MOT body segment is received, the reassembly unit will first check the list provided by the object management. If the *TransportId* of the MOT body segment is described by the list provided by the object management then the MOT body segment will be processed as indicated in the list (i.e. the MOT segment is used for MOT body reassembly or discarded).

If the *TransportId* of the MOT body segment is not mentioned in the list, then the advanced reassembly unit will collect the MOT body segment into the "segment buffer" (hoping that the MOT body segment can be used once the MOT directory is reassembled and forwarded to the object management).

Depending on the available memory in the reassembly unit, more or less MOT body segments can be collected. To collect all MOT body segments received in 5 minutes on a 64 kbps channel one would for instance need 2,4 Mbytes. This number is worst case because it assumes that only MOT body segments with unknown (not in the list provided by the object management) *TransportIds* are received and the current (new) MOT directory can not be reassembled. The advanced reassembly unit will collect as many MOT body segments, as it is able to and may discard the oldest collected MOT body segment in case a newly received MOT body segment needs to be collected and no more memory is available for this purpose (i.e. the advanced reassembly unit must be able to do garbage collection).

The main task for the reassembly unit is the reassembly of MOT bodies. If there is not enough memory to collect MOT body segments in addition to body reassembly, then the reassembly unit will always favour object reassembly.

To assure that no MOT body segments are collected and kept indefinitely, the reassembly unit shall not keep MOT body segments longer than one hour (a timeout is necessary because a content provider might start reusing *TransportIds* after a while). It is therefore necessary to attach a timestamp to all collected MOT body segments.

NOTE: To save memory when collecting MOT body segments it is possible to check if a received MOT body segment was already received and collected before into the "segment buffer". In this case the newly received MOT body segment replaces the earlier received MOT body segment. It might also be necessary to adjust all time stamps attached to this MOT body segment (the time stamps are needed for garbage collection).

C.3.4.2 MOT caching support: relative expire times (MOT parameters Expiration and DefaultExpiration)

If the MOT decoder provides "MOT caching support", then support of the MOT parameters `Expiration` and `DefaultExpiration` is mandatory.

To support relative expire times the MOT reassembly unit needs to know the *TransportId* of the MOT directory currently used by the object management unit. Therefore an interface between the reassembly unit and the object management is required. Whenever the object management accepts a new MOT directory it will indicate the *TransportId* of the currently used MOT directory to the reassembly unit.

Every time the reassembly unit gets an MOT directory segment (data group types 6 or 7) using this indicated *TransportId*, it will store the time the segment was received.

The reassembly unit will update the time when an MOT directory segment using the indicated *TransportId* was received, even if the reassembly unit will then discard all MOT directory segments using the indicated *TransportId* (since the MOT directory was already successfully reassembled).

If persistent caching is used, then the reassembly unit shall permanently store the *TransportId* and the reception time of its last received MOT directory segment. On restart of a persistently cached data application, the reassembly unit shall not assume that *TransportIds* of MOT segments still correspond to the same MOT entity. Therefore it shall NOT update the reception time of the last received MOT directory segment. It shall wait until the object management indicates the *TransportId* of the MOT directory currently in use (i.e. it is necessary to assure that the MOT directory is still the same that it was when the receiver was switched off; to do this the current MOT directory must be successfully reassembled and processed by the object management) before it resumes normal operation (i.e. store the time the last MOT directory segment with the given *TransportId* was received).

See clause C.3.4.3 for a code snippet that outlines the reassembly of the MOT directory; this code also includes some lines to support the handling of relative expire times.

C.3.4.3 Acquiring both compressed and uncompressed MOT directories

If a user application permits both the compressed and the uncompressed MOT directory, then the MOT reassembly unit must be able to reassemble the MOT directory no matter if the MOT directory is compressed or uncompressed or if both the compressed and the uncompressed MOT directory are sent alternately.

An uncompressed MOT directory is carried using MSC data groups of type 6; a compressed MOT directory is carried in data groups of type 7. If both compressed and uncompressed MOT directory are used, then they will have the same *TransportId* for the same version of the MOT directory (i.e. the identical directory content).

One way to acquire the MOT directory (transmitted compressed and/or uncompressed) is to try reassembling both compressed and uncompressed MOT directory in parallel. Reassembly starts whenever a new *TransportId* for the MOT directory is detected (no matter if in a data group of type 6 or 7). Then the parallel reassembly of both the compressed and the uncompressed MOT directory is started using the *TransportId* of the MOT directory. The reassembly unit forwards whatever directory is reassembled first to the object management and stops reassembly of both the compressed and the uncompressed MOT directory until a new *TransportId* of the MOT directory is detected.

The above solution requires enough resources to reassemble both the compressed and the uncompressed MOT directory in parallel. A simpler solution is presented below. It permits to reassemble the MOT directory without the need to decode both the compressed and the uncompressed MOT directory in parallel (i.e. it saves the memory and code lines that would be necessary if the reassembly unit tried to reassemble the compressed and the uncompressed MOT directory in parallel).

Parallel decoding would of course be better (under certain circumstances the MOT directory could be available earlier).

An MOT reassembly unit that just evaluates data group type 6 (uncompressed MOT directory) will contain code something like this pseudo code:

```
// the following variable holds the TransportId of the currently
// reassembled MOT directory
int current_mot_directory_transport_id = -1;

// the following variable is used to know if the MOT directory
// using the above TransportId
// (current_mot_directory_transport_id) is already
// reassembled; in this case all MOT directory segments
// can be ignored
bool mot_directory_completed = false;

// the following variable holds the last time an MOT directory segment
// of the current MOT directory was received
time last_mot_directory_segment_received = 0;

// this function is called whenever a data group of type 6 is received.
process_mot_directory_segment (MOT_segment, unsigned short transport_id)
{
// Support for relative expire times
//
// get_mot_transport_id_used_by_object_management() will return the
// TransportId of the MOT directory currently used by the object management.
// This TransportId does not necessarily have to be the same one as
// current_mot_directory_transport_id since the latter already changes when a
// reassembly of a new MOT directory starts (but reassembly could fail if
// reception is lost).
if (get_mot_transport_id_used_by_object_management() == transport_id) {
// get current time and store it as the last time an MOT segment
// of the currently used MOT directory was received
// last_mot_directory_segment_received = get_current_time();
}

if (current_mot_directory_transport_id != transport_id) {
// we got a new TransportId for the MOT directory, this
// means that we have to rebuild the MOT directory from scratch
restart_mot_directory_reassembly();

// store the now used TransportId
current_mot_directory_transport_id = transport_id;

mot_directory_completed = false;
}

// check if the MOT directory was already reassembled.
// In this case ignore the MOT segment
if (mot_directory_completed) { return; }

// process the currently received MOT directory segment.
add_mot_directory_segment (MOT_segment);

// store information if MOT directory is already completed
mot_directory_completed = is_mot_directory_finished();

// If the MOT directory is now finished, then forward the
// new MOT directory to the reassembly unit
if (mot_directory_completed) {
// forward MOT directory to object management
// ....
}
}

```

To permit reassembly of the compressed or uncompressed MOT directory (carried in data groups of type 6 or 7), the function would now look something like (changes in italic):

```
// the following variable holds the TransportId of the currently
// reassembled MOT directory
int current_mot_directory_transport_id = -1;

// the following variable holds the data group type of the MOT directory
// that is currently reassembled
int current_dg_type;

```

```

// the following variable is used to know if the MOT directory
// using the above TransportId
// (current_mot_directory_transport_id) is already
// reassembled; in this case all MOT directory segments
// can be ignored
bool mot_directory_completed = false;

// the following variable holds the last time an MOT directory segment
// of the current MOT directory was received (data group type 6 or 7!)
time last_mot_directory_segment_received = 0;

// this function is called whenever a data group of type 6 or 7 is received)
process_mot_directory_segment (MOT_segment, unsigned short transport_id,
                              int dg_type)
{
// Handling for relative expire times
//
// get_mot_transport_id_used_by_object_management() will return the
// TransportId of the MOT directory currently used by the object management.
// This TransportId does not necessarily have to be the same one as
// current_mot_directory_transport_id since the latter already changes when a
// reassembly of a new MOT directory starts (but reassembly could fail if
// reception is lost).
if (get_mot_transport_id_used_by_object_management() == transport_id) {
// get current time and store it as the last time an MOT segment
// of the currently used MOT directory was received
last_mot_directory_segment_received = get_current_time();
}

if (current_mot_directory_transport_id != transport_id) {
// we got a new TransportId for the MOT directory,
// this means that we have to rebuild the MOT directory
// from scratch
restart_mot_directory_reassembly();

// store the now used TransportId
current_mot_directory_transport_id = transport_id;
current_dg_type = dg_type;

mot_directory_completed = false;
}

// check if the MOT directory was already reassembled. In
// this case ignore the MOT segment
if (mot_directory_completed) { return; }

// check if the data group type is needed
if (current_dg_type != dg_type) {
// if we first received data group type 6 and now get an
// MOT segment of data group type 7, then we switch to data group type 7.
// If we're already using data group type 7, then we ignore
// data group type 6
if (current_dg_type == 6) {
// we switch to the data group type 7,
// this means that we have to rebuild the MOT directory
// from scratch
restart_mot_directory_reassembly();

// store the now used data group type
current_dg_type = dg_type; // will always be 7
}
else {
// data group type is 6, but we're already reassembling data group type 7
return; // ignore MOT directory segment of type 6
}
}

// process the currently received MOT directory segment.
add_mot_directory_segment (MOT_segment);

// store information if MOT directory is already completed
mot_directory_completed = is_mot_directory_finished();

```

```

// If the MOT directory is now finished, then forward the
// new MOT directory to the reassembly unit
if (mot_directory_completed) {
    // forward MOT directory to object management
    ....
}
}
}

```

The example code starts reassembly with whatever MOT segment is received first (MOT segment with data group types 6 or 7). When the MOT directory is completely reassembled, it will be forwarded to the object management and all further MOT segments with this *TransportId* will be ignored.

If reassembly started with an MOT segment of data group type 6 (i.e. the first MOT segment with a new *TransportId* was from an uncompressed MOT directory) and a MOT segment with data group type 7 is received, then the reassembly unit switches over to the reassembly of the compressed MOT directory (from then on ignoring all MOT segments with data group type 6). If a new *TransportId* for the MOT directory is detected, reassembly starts again with the data group type that is detected first.

This code snippet has no performance penalty (compared to parallel reassembly) if either compressed or uncompressed MOT directory is transmitted. If both compressed and uncompressed MOT directory are used alternately, then the worst-case scenario is a restart of the reassembly of an almost reassembled uncompressed MOT directory with the reception of an MOT segment of data group type 7.

Alternately transmitting compressed and uncompressed MOT directory only makes sense for backwards compatibility to simple receivers that do not support compressed MOT directories. In this case it is sensible to send the compressed MOT directory much more frequently than the uncompressed MOT directory (if the uncompressed MOT directory is sent more frequently, then the MOT directory compression can not sensibly reduce the channel capacity needed for the MOT directory). Therefore switching to data group type 7 should typically result in a quicker reassembly of the MOT directory.

So this simple extension of the reassembly unit will usually have little or no performance penalty compared to parallel reassembly of compressed and uncompressed MOT directory.

C.3.5 Advanced MOT object management

C.3.5.1 MOT directory mode

When the object management unit gets a new MOT directory, it has to compare the old MOT directory with the new MOT directory. The comparison can be done very efficiently if the header information within both the old and the current (new) MOT directory is sorted. In this case two sorted lists have to be compared and this is much more efficient than the comparison of two unsorted lists.

Therefore the advanced MOT object management will sort the header information of the new MOT directory (if not already sorted) before it compares its MOT objects with the (already sorted) old MOT directory.

The presence of the MOT directory extension parameter *SortedHeaderInformation* indicates that the header information is already sorted so that the sorting step can be skipped.

When comparing the sorted MOT header information of the old and the current (new) MOT directory, the compare function defined in annex A shall be used.

After comparing the old and the new MOT directory the object management knows which MOT objects are updated (i.e. those MOT objects with the same *ContentName* in old and current (new) MOT directory whose *TransportId* differs between old and current (new) MOT directory). The MOT decoder will now try to determine if the signalled change to the MOT object also affects the MOT body.

The MOT decoder will check if the MOT object (identified by the same *ContentName*) in both the old and the new MOT directory provides the MOT parameter *UniqueBodyVersion*. If it does and the parameter's value and also the value for *BodySize* is the same in the old and the current (new) MOT directory, then the MOT decoder does not have to rebuild the MOT body; it can use the old (and still valid) version of the MOT body (if already available in the cache). The MOT decoder will nevertheless evaluate the (new) MOT header information (e.g. if an data structure outside the MOT directory is used to keep the expire times of objects then this data structure might have to be updated).

NOTE: When the object management determines that only the object's header information was updated, but it still has the identical body content, it could happen that the object's body is not yet fully reassembled. In this case it is useful if the reassembly unit can be told to finish reassembly of the MOT body even with a different *TransportId*.

Continuing reassembly of an MOT body using a different *TransportId* is not trivial since a different *SegmentationSize* could be used. A good and not too complex compromise seems to be a reassembly unit that is able to continue reassembly if the *SegmentationSize* is the same and that restarts reassembly from scratch if a different *SegmentationSize* for the MOT body is used.

More sophisticated reassembly units will be able to continue reassembly even if a different *SegmentationSize* is used.

The default behaviour of the MOT decoder is to discard all bodies that have been updated (i.e. changed) and wait until the current (new) version of the body is received (thus ignoring the parameter *UniqueBodyVersion*).

The MOT header information parameter *PermitOutdatedVersions* (or if not available for the object, then the MOT directory extension parameter *DefaultPermitOutdatedVersions*) in the current MOT directory tells the MOT decoder if it is permitted to keep the old version of an object (and to provide it to the user application decoder) until the new version of the object is successfully received (see clause 8.1.2.3 for details). As soon as the current (new) version of an object is successfully received, then it has to replace to old one. Note that the user will usually prefer to see older information than nothing at all, so that content providers will most likely permit to temporarily display outdated content. Clause C.3.5.1.1 indicates how an advanced MOT decoder could implement MOT parameters *PermitOutdatedVersions* and *DefaultPermitOutdatedVersions*.

The MOT decoder shall NEVER provide expired MOT objects to the user application decoder. So even if the content provider permits to present an outdated version of an object until the current (new) version of the object is received, it must still be assured that the outdated version is not yet expired!

An MOT decoder that provides "MOT caching support" has to support object expiration (MOT parameters *Expiration* and *DefaultExpiration*). An absolute expire time requires the MOT decoder to check if an object is expired before it honours a request by the user application decoder. A relative expire time requires that the reassembly unit stores the last time when a segment of the currently used MOT directory was received. Therefore the object management has to inform the reassembly unit once it accepts the new MOT directory so that the reassembly unit knows the *TransportId* it should look for.

If the user application decoder requests an MOT object, then the object management will thus first check if an expire time is given for the object.

If an absolute expire time has been given, then the object is considered expired if the user application's request was issued after the given absolute expire time.

If a relative expire time has been given, the object management will ask the reassembly unit when a segment of the current MOT directory was most recently received. The object is considered expired if the user application's request was issued after this time plus the given relative expire time.

If the MOT object is requested and it is not expired, then the object management will forward the object to the user application decoder.

Even though the MOT decoder shall never provide expired objects to the user application decoder it might sometimes be sensible to keep MOT objects in the cache even if they already expired.

Imagine a scenario where the MOT decoder collected some HTML data and the user wants to browse through it in the underground. Some traffic data might have a timeout of just some minutes (using relative expire times). Since the MOT decoder might not receive anything in the underground, it will not provide the expired traffic information to the user application decoder after the timeout expired. Once the user gets back to the surface and reception of the HTML data continues, the MOT decoder will (after receiving the MOT directory) detect which of the traffic messages are still valid (i.e. which are still broadcast). Deleting all MOT objects once they expired would have delayed presentation of the data once reception is available again. (Note that if the receiver was switched off, then the MOT decoder also has to support the MOT parameter *UniqueBodyVersion* and that this parameter must also be included in the MOT directory: once reception is restored, the parameter *UniqueBodyVersion* will reliably tell the MOT decoder which MOT objects are still valid).

The MOT directory contains object descriptions of all currently broadcast objects and their *TransportIds*, but it does not signal when which body is broadcast. Therefore an object management strategy that reassembles objects with the highest priority first might have a bad start-up time since all other objects broadcast before these objects are ignored. The reassembly unit might signal to the object management which body *TransportIds* are currently broadcast. E.g. if the first segment (not necessarily *SegmentNumber* 0) of any body is received, this could be signalled to the object management unit. The object management could then order the reassembly of this body.

The time when the first body segment is received and the *RetransmissionDistance* parameter of the object can be used by the object management unit to predict the time of the next retransmission of the body. This permits advanced caching strategies.

It is also possible to evaluate the data group *Repetition* index (see clause 5.3.3.1 in EN 300 401 [1]) or the *RepetitionCount* of the segmentation header (see clause 5.1.1).

However, such additional processing by the reassembly unit and object management unit is just optional.

C.3.5.1.1 Support of MOT parameters *DefaultPermitOutdatedVersions* and *PermitOutdatedVersions*

If the content provider indicates that an older (and not yet expired) version of an MOT object can be used until the current (new) version of an MOT object is received, then it is necessary to keep the entire MOT object (i.e. MOT header information as well as the MOT body) until the current (new) version of the MOT object is successfully reassembled.

One way to achieve this is a data structure that holds all MOT objects that later on will be replaced by the current (new) version of the MOT object.

Initially this data structure will be empty. Every time an update to the MOT directory is made, the MOT decoder will check all current (new) MOT objects.

If an update to the MOT object body is signalled, then the MOT decoder will check if it is permitted to use older versions of this MOT object. If it is permitted and the MOT object is available in the cache (i.e. the MOT body is completely reassembled), then this object is added to the data structure.

If an MOT object within the data structure is no longer signalled in the MOT directory, then it will be removed from the data structure.

Every time an MOT object (i.e. its body) is successfully reassembled, the MOT decoder will check if an older version of this MOT object is in the above data structure. If it is, the old version of the MOT object will be removed from the above data structure.

If the MOT decoder is requested to provide an MOT object, then it will always first check if the MOT object is in the above data structure. If it is, then the MOT decoder will check if the (old) MOT object has already expired. If it is expired, the MOT decoder will remove the MOT object from the above data structure. If the object is still valid it will be forwarded to the user application decoder.

Garbage collection can regularly remove expired MOT objects from the above data structure.

An old (outdated) object that is added to the above data structure needs special handling of relative expire times. The relative expire time indicates how many minutes the MOT objects should be considered valid after reception loss (or more precise: after the validity of the MOT object can no longer be confirmed).

If the MOT parameter *PermitOutdatedVersions/DefaultPermitOutdatedVersions* signals that an old (outdated) version of an MOT object may be used while the current version is reassembled, then the relative expire time therefore indicates how many minutes the old (outdated) object might still be used at most.

MOT demands that from the instant in time a new version of an object is sent, the old version will no longer be sent. Therefore it is very easy to determine the absolute expire time of an outdated MOT object if a relative expire time is indicated for the object. The MOT object will expire that many minutes (as indicated by the relative expire time) after the current (new) MOT directory (that caused the MOT object to be considered "outdated") was successfully reassembled.

So if an MOT object is added to the above data structure, then the absolute expire time for this MOT object is determined by taking into account relative or absolute expire time of this object.

The MOT object management "keeps" an entire outdated object (body and header information), but in MOT directory mode a user application might also put user application specific MOT parameters into the MOT directory extension. These user application specific parameters are only valid and available as long as the MOT directory carrying them is available. They are *not* related to any particular version of any individual object. Therefore content providers shall only permit "keeping" of an outdated version of an object, if no inconsistency between the information carried in the outdated (old) MOT header information and the current (new) MOT directory can occur.

An inconsistency could for instance occur if the user application uses parameters in both MOT header information and MOT directory extension to efficiently code some data.

An example is the use of a default parameter value (carried in the MOT directory extension) that applies to all MOT objects unless a parameter in the MOT header information of an object explicitly sets a different value for this object. Another example is a MOT directory extension parameter carrying a mapping table that permits to use short "keys" in the MOT header information instead of long parameter values. The full (long) parameter value is determined by the user application decoder by a look-up in the MOT directory extension parameter that maps the short key to the final parameter value used by the user application decoder.

In both examples an inconsistency could occur if the data in the MOT directory extension is changed and an outdated object is "kept". Therefore user applications that use this or a similar memory optimization have to take care to address this potential problem.

C.4 User application level

The user application level requests objects from the MOT decoder (MOT directory mode) or gets every object as soon as it is successfully received (MOT header mode) and presents them.

The specification of the user application level is not a part of MOT.

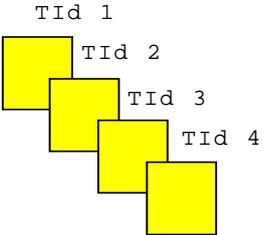
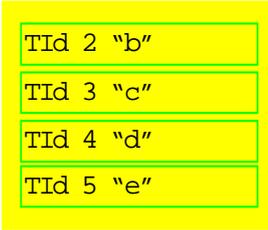
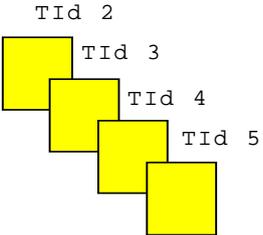
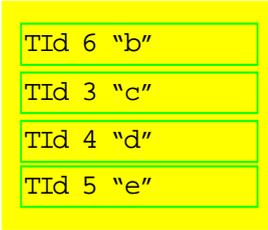
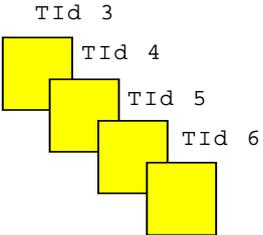
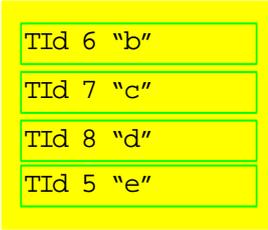
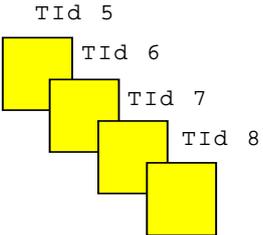
The MOT decoder shall NEVER provide expired MOT objects to the user application decoder.

Annex D (informative): MOT decoding in MOT directory mode (example)

This example shows the actions of the MOT decoder after it receives an MOT directory. The MOT decoder starts from scratch (i.e. there are no objects already available).

TId is used as an abbreviation for *TransportId*. If the same TId appears more than once in the example, it is the same object, i.e. the same header and body.

Table D.1

MOT directory	Actions to do after reception of the MOT directory	Objects in the object management
<p>TId 100</p> 	<ul style="list-style-type: none"> ▪ store objects with TId 1, 2, 3, 4 	
<p>TId 101</p> 	<ul style="list-style-type: none"> ▪ delete object with TId 1 ▪ keep objects with TId 2, 3, 4 ▪ store object with TId 5 	
<p>TId 102</p> 	<ul style="list-style-type: none"> ▪ delete object with TId 2 ▪ keep objects with TId 3, 4, 5 ▪ store object with TId 6 	
<p>TId 103</p> 	<ul style="list-style-type: none"> ▪ delete objects with TId 3, 4 ▪ keep objects with TId 6, 5 ▪ store objects with TId 7, 8 	

Annex E (informative): Example for evaluation of relative expire times (MOT parameters Expiration and DefaultExpiration)

In this example the following assumptions are made:

- At the beginning the *TransportId* of the current MOT directory is 0x1000.
- All requests by the user application decoder refer to the same MOT object. This object is already available in the cache at 10:00:00.
- The relative expire time given for the requested MOT object is 15 minutes.
- Starting at 10:02:00 a new MOT directory with *TransportId* 0x1001 using 2 MOT directory segments is received. The change in the MOT directory does not affect the MOT object requested by the user application (it still uses the same *TransportId* in the new MOT directory). Not before 10:18:00 the MOT decoder successfully reassembles the new MOT directory.
- There is no reception between 10:02:01 and 10:17:59 and after 10:19:00.

Table E.1

Time	MOT directory segment reception	Object-request by user application decoder
10:00:00	An MOT directory segment with <i>TransportId</i> 0x1000 is received. Set time of last received MOT directory segment to 10:00:00.	
10:00:10		MOT object will expire at 10:00:00 + 15 minutes (10:15:00); it is still valid now. Return requested MOT object.
10:00:30	An MOT directory segment with <i>TransportId</i> 0x1000 is received. Set time of last received MOT directory segment to 10:00:30.	
10:00:45		MOT object will expire at 10:00:30 + 15 minutes (10:15:30); it is still valid now. Return requested MOT object.
10:02:00	An MOT directory segment with <i>TransportId</i> 0x1001 is received. MOT directory with <i>TransportId</i> 0x1001 is not yet completely reassembled. The currently used MOT directory uses <i>TransportId</i> 0x1000. No change to the time of last received MOT directory segment.	
10:05:20		MOT object will expire at 10:00:30 + 15 minutes (10:15:30); it is still valid now. Return requested MOT object.
10:16:00		MOT object expires at 10:00:30 + 15 minutes (10:15:30); it is no longer valid now and therefore not returned (but probably still kept in the cache until validity of the object can later be re-confirmed) Indicate that object is not available.
10:18:00	An MOT directory segment with <i>TransportId</i> 0x1001 is received. MOT directory with <i>TransportId</i> 0x1001 is now completely reassembled and now used as the current MOT directory. Set time of last received MOT directory segment to 10:18:00.	
10:20:00		MOT object will expire at 10:18:00 + 15 minutes (10:33:00); it is still valid now. Return requested MOT object.
10:35:00		MOT object expires at 10:18:00 + 15 minutes (10:33:00); it is no longer valid now and therefore not returned (but probably still kept in the cache until validity of the object can later be re-confirmed) Indicate that object is not available.

Annex F (informative): Managing changes to the MOT data carousel

The followings clauses outline how the content provider will manage changes to his data carousel.

F.1 General principle

The MOT directory is usually periodically transmitted in parallel to the MOT bodies. The bitrate of the MOT directory will usually be some predefined percentage of the total bitrate used by the content provider.

On start-up of the MOT decoder no MOT object can be reassembled before the MOT directory is successfully reassembled and processed for the first time. Therefore the cycle time between two retransmissions of the MOT directory is an important factor to determine how fast data can be accessed on start-up of the MOT decoder and how fast changes to the data carousel propagate to the MOT decoder. To assure that the MOT directory can be sent frequently enough without requiring too much channel capacity, the MOT directory should be made as small as possible (e.g. by using short `ContentNames`) and MOT directory compression should be used (if permitted by the user application definition).

If a change to the set of broadcast objects is made, then the content provider will stop sending the old MOT directory and he will also stop sending MOT segments of MOT bodies that are no longer part of the updated data carousel.

He will then set up the new MOT directory describing the current state of the data carousel and send this new MOT directory (with a new *TransportId*) for some time (to "assure" reception). Then he will start sending the new and updated MOT bodies (in parallel to the MOT directory). After the new and updated objects were sent, the content provider will continue sending the unchanged MOT objects.

It is important that a change to the data carousel does not completely restart the transmission cycle. To give an example:

- If the data carousel cycle is five minutes and some small object is changed every minute, then the data carousel shall not restart from scratch and shall not start repeating the first minute of the data carousel without ever broadcasting the data scheduled for the remaining four minutes.

It is also important to assure that the *TransportId* of MOT bodies is only changed if their header information, body content or segmentation is changed. Every unnecessary change of the *TransportId* will cause simple MOT decoders to discard all the already received segments of the MOT body!

The MOT parameter `UniqueBodyVersion` should be provided so that the MOT decoder can easily determine if a change of an object's *TransportId* also affected the MOT body. Whenever possible, the content provider should also aim at permitting to keep outdated objects on receiver side (MOT parameters `PermitOutdatedVersions` and `DefaultPermitOutdatedVersions`).

F.2 Advanced approach

In the above scenario the MOT directory will be sent quite often in case of a change to the data carousel. The repetition of the MOT directory is necessary to assure that every MOT decoder will know the *TransportIds* of the new and updated MOT objects. Simple MOT decoders will discard all MOT segments whose *TransportId* they do not (yet) know.

Advanced MOT decoders collect MOT segments of MOT bodies whose *TransportIds* are not (yet) listed in the current MOT directory (see clause C.3.4.1). Therefore the content provider can make faster changes to the data carousel and then he needs less bitrate for the MOT directory.

In this case the content provider can send the new MOT directory in parallel to the new and updated MOT bodies. This way the bitrate demand of the MOT directory will not increase with the update frequency of the data carousel. The content provider will output MOT directory segments in regular intervals (depending on the bitrate assigned to the MOT directory). If a change to the data carousel is made, then segments of the new MOT directory are output; it is not necessary to increase the bitrate of the MOT directory.

The above approach has the following implications:

- MOT decoders that collect MOT body segments with unknown *TransportId* in a "segment buffer" will be able to use the previously collected MOT body segments as soon as the current (new) MOT directory is received. However, since the MOT directory is needed to identify the MOT object these MOT body segments belong to, the MOT decoder has to wait until the MOT directory is received and the collected MOT body segments are "replayed" by the reassembly unit, before the MOT decoder can identify and forward a new or updated object to the user application decoder.
- MOT decoders that do NOT collect MOT body segments with unknown *TransportId* will most likely miss many of the new and updated MOT bodies on their first turn of the data carousel. They will probably need to wait an additional turn of the data carousel.
- The content provider has to assure that every version of the MOT directory is sent often enough so that it can be reassembled even under bad reception condition. It is possible to repeat an "old" MOT directory in parallel to new and updated (and unchanged) MOT bodies until a certain reception probability for the MOT directory has been reached. Then all changes to the data carousel (no matter if there was one big change or many small ones) can be combined in a new MOT directory that is then again sent for some time.

Note that collecting MOT body segments in a "segment buffer" not only permit faster changes to the data carousel. It also provides better performance on MOT decoder startup and in case of reception errors during a change to the data carousel (i.e. if the current (new) MOT directory could not be reassembled on the first transmission).

User applications that foresee fast changes to the data carousel should require collecting ("segment buffer") of MOT body segments whose *TransportIds* are not (yet) listed in the current MOT directory (see clause C.3.4.1).

Annex G (informative): Implementation tips for "Transfer of directory structures via MOT" (receiver side)

The MOT Decoder can make use of (at least) two strategies that both permit to access MOT objects inside a directory structure:

- The first strategy uses the "generic" MOT decoder (see clause 6.2.2.1.1) and provides an additional interface for user applications that use the "transfer of directory structures".

The objects are stored in the cache using an internal filename (for example built using the `TransportId` of the object). The current mapping between the internal filenames and the corresponding `ContentNames` is also stored. When the user application decoder requests an object with a certain `ContentName`, the interface to the MOT decoder uses the mapping list to determine and return the corresponding internal filename. The advantage of this method is that the MOT decoder does not have to check the validity of the `ContentName` as a pathname on the file system and does not have to create or destroy directories when storing or deleting MOT objects. The disadvantage is that the mapping list has to be maintained and that all access from the user application to the objects must be done via a special interface.

NOTE 1: To simplify access it is possible to provide a virtual file system. In this case there would be a special interface between the MOT decoder and the user application that provides a virtual file system. A possible solution would be an NFS (network file system) or an SMB server. It would provide a file system with the full directory structure that can be "mounted". If for instance the MOT carousel is then "mounted" as drive "m:" on a Windows system, then the user application could work on this drive "m:" as if it were a local file system. But all file access operations will pass through the interface to the MOT decoder and thus the MOT decoder will have full knowledge of what files are accessed by the user application. It will also be possible to "export" the received data to other systems (e.g., via a WLAN network).

The second strategy defines an additional operation mode for the MOT decoder. This mode of the MOT decoder uses a different way to determine the file name that is used to store an MOT object.

If the user application decoder tells the MOT decoder that "transfer of directory structures" is used, then the MOT objects are stored using the `ContentName` as a filename relative to a directory on the file system of the receiver (for instance, let's assume that the MOT decoder stores all files in its cache directory "`\mot\service_d312\sc_1\`". A file with `ContentName` "`root/dir1/file2`") would then be stored as "`\mot\service_d312\sc_1\root\dir1\file2`"): the advantage is that no mapping list has to be maintained and the files can directly be accessed in the file system of the receiver. The disadvantage is that the MOT decoder has to create or destroy directories and check that the given `ContentName` is a valid pathname on the file system. The MOT decoder also gets no feedback about which objects were accessed at what time; information that could be useful for the cache management.

The user application decoder just needs to know the path to the cache of the MOT decoder to find the selected objects by concatenating the directory name of the MOT decoder's cache and the (relative) file name the object given in the `ContentName` (i.e., the full file name would then be "`<Cache path>/<ContentName>`").

NOTE 2: An MOT decoder that uses this approach (i.e., the user application directly accesses files in the MOT decoder's cache) must be aware that on some operating systems it is impossible to update (replace) or delete a file if this file is currently in use by the application decoder. This means that the MOT decoder may not assume that replacing a file with its updated version or removing an obsolete file will always succeed. The MOT decoder will need to keep a list of all necessary file replacements/file and directory deletions that could not be performed because these files are currently accessed by the user application decoder. Every now and then the MOT decoder must then check if the file operations can be performed. If some files are often updated/deleted, then multiple entries per file may be added to the list.

NOTE 3: This strategy can be implemented in two ways: either the MOT decoder directly stores the files according to its `ContentName`, or an special interface (e.g., an additional program) automatically manages the directory structure by creating and deleting all files and directories. The first approach means two operating modes for the MOT decoder ("generic" mode and "directory transfer" mode), the latter approach requires an additional entity that connects to the "generic" MOT decoder and builds the directory structure.

Annex H (informative): Bibliography

- ISO/IEC 7498 (all parts): "Information technology - Open Systems Interconnection - Basic Reference Model".
- IETF RFC 1950: "ZLIB Compressed Data Format Specification version 3.3".
- IETF RFC 1952: "GZIP file format specification version 4.3".
- CENELEC EN 62106: "Specification of the radio data system (RDS) for VHF/FM sound broadcasting in the frequency range from 87,5 to 108,0 MHz".

History

Document history			
V1.1.1	January 1998	Publication	
V1.2.1	February 1999	Publication	
V2.1.1	February 2005	One-step Approval Procedure (Withdrawn)	OAP 20050603: 2005-02-02 to 2005-06-03
V2.1.1	January 2006	One-step Approval Procedure	OAP 20060519: 2006-01-18 to 2006-05-19
V2.1.1	May 2006	Publication	